计 算 机 基 础 课 程 系 列 教 材

# C++程序设计教程

王珊珊 臧洌 张志航 编著

机械工业出版社
China Machine Press

计算机基础课程
系列教材

# C++程序设计教程

本书凝聚了作者十余年的C/C++程序设计语言教学经验。其间又历经两年的试用，几易其稿，最终才得以面世。全书详细讲解了C++程序设计的基础知识，注重培养读者面向对象的实际编程能力。

## 本书特色

- 注重对读者算法思路的逻辑训练。
- 针对教学重点和难点，配有丰富的例题和习题。初学者在学习完相关章节的内容后，拿到题目，即能够独立编程，而不至于无从下手。
- 本书配有如下教学资料：本书全部例题的源代码、全部习题的解答、各章教学所用的PowerPoint讲稿。如果教师选用本书做教材，则可登录华章网站获取上述资料。

上架指导：计算机/程序设计

ISBN 7-111-17368-6
9 787111 173687
封面设计：关 刚

HZ BOOKS
华章图书

华章网站 http://www.hzbook.com
网上购书：www.china-pub.com
投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com
ISBN 7-111-17368-6/TP · 4454 (课)
定价：33.00 元

计 算 机 基 础 课 程 系 列 教 材

# C++程序设计教程

王珊珊 臧洌 张志航 编著

机械工业出版社
China Machine Press

C++语言是目前被广泛采用的程序设计语言，它语法简洁，运行高效，既可以进行面向过程的程序设计，又可以进行面向对象的程序设计。因此，它既适于程序设计语言的初学者，又适于从面向过程程序设计技术过渡到面向对象程序设计技术的学习者阅读。

本书全面讲解面向过程的程序设计及面向对象程序设计的基本概念。其间针对教学重点和难点，配有大量的例题，使初学者学习完相关章节的内容后，拿到题目，即能够独立编程，而不致茫然无措。

本书适于大学本科非计算机理工类各专业师生阅读，对计算机专业的学生来说也是一本较好的参考书，同时适于自学C++语言的读者阅读。

**版权所有，侵权必究。**
**本书法律顾问　北京市展达律师事务所**

**图书在版编目（CIP）数据**

C++程序设计教程 / 王珊珊等编著. - 北京：机械工业出版社，2006.1
（计算机基础课程系列教材）
ISBN 7-111-17368-6

Ⅰ. C…　Ⅱ. 王…　Ⅲ. C语言 - 程序设计 - 教材　Ⅳ. TP312

中国版本图书馆CIP数据核字（2005）第105811号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）
策划编辑：王子恢
责任编辑：李云静
北京瑞德印刷有限公司印刷 · 新华书店北京发行所发行
2006年1月第1版第1次印刷
787mm × 1092mm 1/16 · 22.75印张
印数：0 001-4 000册
定价：33.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

# 前　言

## 编写背景

教育部高教司早在“八五”期间已经把计算机基础教学纳入高校的重要基础课程中，提出了“加强工科非计算机专业计算机基础教学工作的几点意见”，对计算机基础教学的地位和任务（培养目标）、课程体系和教学内容、教学方法和手段、师资和教材建设以及基础硬件设施的建设等多方面提出了指导性的意见。

其中最重要的一点是提出了计算机基础教学的三层次教学模式（即计算机文化基础、计算机技术基础和计算机应用基础）和课程体系。高级程序设计语言类课程属于课程体系的第二个层次。

经过多年的教学实践，教育部高教司于2003年又提出了“进一步加强高校计算机基础教学的几点意见（征求意见稿）”（俗称计算机基础教育白皮书），针对计算机基础教学的现状与发展，提出了计算机基础教学改革的指导思想；按照分类、分层次组织教学的思路，提出了计算机基础课教学内容的知识结构与课程设置。在附件中给出了六门计算机基础教学的典型核心课程，并给出了这六门课程的教学大纲。这六门课程包括：大学计算机基础、计算机程序设计基础（含C和C++）、计算机硬件技术基础（微机原理与接口、单片机原理与应用）、数据库技术与应用、多媒体技术与应用、网络技术与应用。

另外，全国高校计算机基础教育研究会成立了“高等院校计算机基础教育改革课题研究组”。该研究组于2004年出版了课题报告《中国高等院校计算机基础教育课程体系2004》（简称CFC 2004），对国内计算机基础教育进行了全面的总结和系统研究，是指导计算机教育改革的重要文件。

本书正是依据上述“白皮书”中给出的“计算机程序设计基础（C++）”的大纲编写而成，同时也满足CFC 2004中所给的C++语言的主要内容。

## 编写内容和教学要求

目前，各大专院校非计算机专业，尤其是电类专业，普遍选用程序设计语言C++作为程序设计教学的第一语言。为什么要选用面向对象的程序设计语言C++呢？因为目前市场上绝大部分通用的计算机软件开发工具都是基于面向对象的编程工具，如Visual C++、Delphi、Java和C# .NET等。随着计算机应用的普及，迫切需要大专院校学生具备面向对象的编程能力才能够适应社会的需求。本书就是为了满足这个层次的学生需求而编写的。本书内容是使用面向对象通用软件开发工具的基础。

本书第1~9章是结合C++语言的基本语法，介绍传统的面向过程的程序设计。内容包括C++语言基本数据类型、基本运算、基本输入/输出、结构化流程控制语句、函数、编译预处理、数组、结构体、指针和链表等内容，基本上是传统的C程序设计语言的内容。第二部分第10～15章是面向对象的程序设计，以面向对象的三大特点为主线，讲述类和对象的基本概念，类的封装、继承和多态，函数模板和类模板，并讲述了输入/输出流类体系、文件操作等内容。

本书作者主张的教学理念是注重程序设计算法的教学，注重对学生算法思路的逻辑训练，而不拘泥于语法要素的细枝末节。本书通俗易懂，各章节配有大量的例题和习题，主要是针对各章的教学难点和重点以及各种算法而设计的。在选择例题和习题时，尽量涵盖目前程序设计语言课程的各种算法类型。使初学者拿到习题后，能够在本书例题中找到相似的例子，这样对于初学者来说，解题就不是一件非常困难的事情了。除了进行课本教学外，教师还可以根据实际情况选择安排课程设计，选用适合不同层次学生的课程设计题目，加强学生动手编制较大规模程序的能力。有关课程设计的书可参考机械工业出版社出版的《C++

程序设计课程设计》（ISBN: 7-111-14565-8）。

本书第1、2、9~14章由王珊珊编写，第5~8章由臧冽编写，第3、4、15章由张志航编写，全书由王珊珊负责统稿。

讲述本书全部内容的建议学时为：理论教学56学时，课程设计16学时（内容另行安排），上机实验60学时。本书的实验环境是Visual C++ 6.0，本书中全部的例题和习题均在该环境中通过编译和运行。

本书可能会存在疏漏、不妥和错误之处，恳请广大读者指正。

## 联系方法

作者电子邮件地址:
shshwang@nuaa.edu.cn（王珊珊）
coralwang2002@yahoo.com. cn（王珊珊）
zangliwen@yahoo.com.cn（臧冽）
zqwzzh@nuaa.edu.cn（张志航）

作 者
2005年10月20日
于南京航空航天大学

# 目　录

前言

## 第一部分　面向过程的程序设计

第1章　C++概述……1
1.1　计算机语言与程序……1
1.1.1　机器语言与程序……1
1.1.2　汇编语言与程序……1
1.1.3　高级语言与程序……2
1.2　从C到C++……2
1.3　程序设计方法……3
1.3.1　结构化程序设计方法……3
1.3.2　面向对象的程序设计方法……3
1.4　简单的C++程序介绍……4
1.5　程序开发的步骤……5
习题1……5
第2章　数据类型、运算符和表达式……7
2.1　保留字和标识符……7
2.1.1　保留字……7
2.1.2　标识符……7
2.2　C++的基本数据类型……7
2.3　常量和变量……9
2.3.1　常量……9
2.3.2　符号常量……10
2.3.3　变量……11
2.4　基本运算符和表达式……11
2.4.1　C++运算符及表达式简介……11
2.4.2　算术运算符和算术表达式……12
2.4.3　运算优先级和结合性……12
2.4.4　关系运算符和关系表达式……13
2.4.5　逻辑运算符和逻辑表达式……13
2.4.6　位运算符和位运算表达式……13
2.4.7　自增、自减运算符和表达式……15
2.4.8　赋值运算符和赋值表达式……15
2.4.9　逗号运算符和逗号表达式……16
2.4.10　sizeof( ) 运算符和表达式……16
2.4.11　逻辑表达式运算优化时的副作用…16
2.5　类型转换……17
2.5.1　赋值时的自动类型转换……17
2.5.2　各种类型运算量混合运算时的自动类型转换……18
2.5.3　强制类型转换……19
习题2……19
第3章　简单的输入／输出……20
3.1　传统的输入／输出函数实现方法……20
3.2　cin……21
3.2.1　输入十六进制或者八进制数据……23
3.2.2　输入字符数据……24
3.3　cout……26
3.3.1　输出八进制数、十六进制数和用科学计数法表示的数……28
3.3.2　输出字符或字符串……29
3.4　总结……29
习题3……30
第4章　C++的流程控制……31
4.1　算法概述……31
4.1.1　算法的作用和类别……31
4.1.2　算法的设计原则……32
4.1.3　算法的表示工具……33
4.1.4　结构化程序设计中基本结构的表示……34
4.2　C++程序的结构和语句……36
4.3　选择结构语句的使用……38
4.3.1　if语句……38
4.3.2　if语句的嵌套使用……41
4.3.3　条件运算符……42
4.3.4　开关语句……43
4.4　循环结构语句的使用……46
4.4.1　goto语句及标号的使用……46

4.4.2 while语句 ……47
4.4.3 for语句 ……47
4.4.4 do-while语句 ……49
4.4.5 break语句和continue语句 ……50
4.4.6 循环的嵌套 ……51
4.5 控制语句的应用举例 ……52
习题4 ……57
第5章 函数 ……58
5.1 概述 ……58
5.2 函数的定义与调用 ……58
5.2.1 函数的定义 ……58
5.2.2 函数的调用 ……60
5.2.3 函数的参数 ……62
5.3 函数的原型说明 ……63
5.4 函数的嵌套调用和递归调用 ……64
5.4.1 函数的嵌套调用 ……64
5.4.2 函数的递归调用 ……68
5.5 内联函数 ……71
5.6 函数重载 ……72
5.6.1 参数类型不同的重载函数 ……72
5.6.2 参数个数不同的重载函数 ……72
5.7 使用C++系统函数 ……73
5.8 作用域和存储类别 ……74
5.8.1 作用域 ……74
5.8.2 存储类别 ……77
5.9 程序的多文件组织 ……82
5.9.1 内部函数和外部函数 ……82
5.9.2 多文件组织的编译和连接 ……83
习题5 ……84
第6章 编译预处理 ……85
6.1 宏定义 ……85
6.1.1 不带参数的宏定义 ……85
6.1.2 带参数的宏定义 ……87
6.2 “文件包含”处理 ……89
*6.3 条件编译⊖ ……90
习题6 ……92
第7章 数组 ……93
7.1 数组的定义及应用 ……93
7.1.1 一维数组的定义及使用 ……93
7.1.2 一维数组做函数参数 ……96
7.1.3 多维数组的定义及使用 ……105
7.1.4 二维数组名用做函数参数 ……107
7.2 字符数组的定义及应用 ……111
7.2.1 字符数组的定义 ……111
7.2.2 字符数组的初始化 ……112
7.2.3 字符数组的使用 ……112
7.2.4 字符串和字符串结束标志 ……112
7.2.5 字符数组的输入/输出 ……113
7.2.6 字符串处理函数 ……115
7.2.7 字符数组应用举例 ……117
习题7 ……119
第8章 结构体、共用体和枚举类型 ……121
8.1 结构体的定义及应用 ……121
8.1.1 结构体类型的定义 ……121
8.1.2 结构体类型变量的定义 ……122
8.1.3 结构体类型变量及其成员的引用 ……123
8.1.4 结构体数组 ……126
8.2 共用体的定义及应用 ……129
8.2.1 共用体类型及其变量的定义 ……129
8.2.2 共用体类型变量的引用 ……129
8.2.3 共用体数据类型的特点 ……130
8.3 枚举类型 ……131
8.3.1 枚举类型的定义 ……131
8.3.2 枚举类型变量的定义 ……131
8.3.3 枚举类型变量的使用 ……131
习题8 ……133
第9章 指针和引用 ……135
9.1 指针和指针变量 ……135
9.1.1 指针的概念 ……135
9.1.2 指针变量的定义 ……135
9.1.3 有关指针的运算符 & 和 * ……135
9.1.4 指针变量的初始化 ……136
9.1.5 直接访问和间接访问 ……136
9.1.6 地址值的输出 ……138
9.2 指针做函数参数 ……139
9.2.1 基本类型量做函数参数 ……139

⊖ 章节编号或例题编号等前有星号“*”者为较难理解的内容或选学内容。——编者注

9.2.2 指针变量做函数参数 ……140
9.2.3 从被调函数中返回信息的几种方法…141
9.3 指针和指向数组的指针 ……142
9.3.1 指针和一维数组 ……142
9.3.2 一维数组元素指针做函数参数 ……145
9.3.3 指针和字符串 ……148
9.3.4 多维数组指针（二维数组） ……151
9.4 指针数组 ……155
9.4.1 指针数组的定义和使用 ……155
9.4.2 使用指针数组处理二维数组 ……155
9.4.3 利用字符指针数组处理字符串 ……156
9.4.4 指针数组做 main 函数的参数 ……158
9.5 指向指针的指针 ……159
9.6 指针和函数 ……159
9.6.1 函数指针 ……159
9.6.2 返回指针值的函数(指针函数) ……162
9.7 指针小结 ……162
9.8 引用和其他类型的指针 ……163
9.8.1 引用类型变量的说明及使用 ……163
9.8.2 引用和函数 ……164
9.8.3 const类型量 ……165
9.9 存储空间的动态分配和释放 ……167
9.9.1 new和delete运算符 ……167
9.9.2 使用new和delete运算符的注意事项 ……168
9.10 链表及其应用 ……169
9.10.1 结构体及指针 ……169
9.10.2 链表概念的引入 ……170
9.10.3 有关链表的算法 ……171
习题9 ……176

## 第二部分 面向对象的程序设计

第10章 类和对象 ……179
10.1 类和对象的定义 ……179
10.1.1 从结构体到类 ……179
10.1.2 类和对象的定义形式 ……179
10.1.3 对象成员的访问 ……180
10.1.4 成员函数的定义 ……181
10.1.5 对象的存储空间 ……183
10.1.6 定义类和对象的有关说明 ……183
10.2 初始化对象、撤销对象 ……185
10.2.1 构造函数和析构函数 ……185
10.2.2 缺省构造函数和缺省析构函数 …190
10.2.3 拷贝构造函数和缺省拷贝构造函数 ……191
10.2.4 拷贝构造函数的调用时机 ……193
10.2.5 利用构造函数进行类型转换 ……195
10.3 成员函数的特性 ……196
10.3.1 内联函数和外联函数 ……196
10.3.2 成员函数的重载 ……197
10.4 构造函数和对象成员 ……199
10.5 this指针 ……201
习题10 ……202
第11章 类和对象的其他特性 ……211
11.1 静态成员 ……211
11.1.1 静态数据成员 ……211
11.1.2 静态成员函数 ……212
11.2 友元 ……213
11.2.1 友元函数 ……213
11.2.2 一个类的成员函数可作为另一个类的友元函数 ……215
11.2.3 友元类 ……216
习题11 ……216
第12章 继承和派生 ……218
12.1 继承的基本概念 ……218
12.2 单一继承 ……218
12.2.1 公有继承(派生) ……219
12.2.2 私有继承(派生) ……222
12.2.3 保护继承(派生) ……223
12.2.4 private成员和protected成员的区别 ……223
12.3 多重继承 ……223
12.4 基类成员的初始化 ……225
12.4.1 基类的构造函数和析构函数的调用顺序 ……225
12.4.2 对象成员构造函数和析构函数的调用顺序 ……226
12.5 二义性和支配规则 ……228
12.5.1 二义性（访问冲突） ……228
12.5.2 支配规则 ……231

12.6 虚基类 ……232
12.7 访问基类成员和访问对象成员的成员 ……235
12.7.1 访问对象成员的成员 ……235
12.7.2 访问基类成员 ……236
12.8 赋值兼容 ……237
习题12 ……238
第13章 多态性 ……244
13.1 函数重载 ……244
13.2 运算符重载 ……245
13.2.1 运算符重载的几点说明 ……245
13.2.2 运算符重载的两种方式 ……246
13.2.3 类型转换函数——将本类对象转换成其他类对象 ……258
13.2.4 其他运算符的重载 ……259
13.2.5 字符串类 ……269
13.3 静态联编 ……275
13.4 动态联编和虚函数 ……276
13.4.1 虚函数的定义和使用 ……276
13.4.2 虚析构函数 ……280
13.5 纯虚函数和抽象类 ……282
习题13 ……290
第14章 输入/输出流 ……296
14.1 输入/输出基本概念 ……296
14.1.1 输入/输出流 ……296
14.1.2 文本流、二进制流和数据文件 ……296
14.1.3 缓冲 ……296
14.2 输入/输出流类体系 ……297
14.2.1 基本输入/输出流类体系 ……297
14.2.2 用运算符重载实现标准设备的输入/输出 ……298
14.2.3 缺省的输入/输出格式 ……299
14.3 输入/输出格式控制 ……301
14.3.1 使用成员函数进行格式控制 ……301
14.3.2 使用操纵算子进行格式控制 ……305
14.4 使用成员函数实现输入/输出 ……307
14.4.1 输出函数 ……308
14.4.2 输入函数 ……308
14.5 重载插入和提取运算符 ……311
14.6 文件流类 ……315
14.6.1 文件流类体系 ……315
14.6.2 文件的打开和关闭 ……315
14.6.3 文本文件的读写 ……318
14.6.4 二进制文件的读写 ……323
14.7 文件的随机访问 ……325
14.8 输入/输出流的出错处理 ……327
习题14 ……329
第15章 模板 ……331
15.1 概述 ……331
15.2 函数模板 ……332
15.2.1 定义函数模板 ……333
15.2.2 模板函数的重载 ……336
15.2.3 函数模板的重载 ……338
15.2.4 多重数据类型参数的模板函数 ……339
15.3 类模板 ……341
15.3.1 定义类模板 ……341
15.3.2 类模板的友元函数 ……346
15.3.3 类模板的特殊处理 ……349
15.4 总结 ……350
习题15 ……351
附录A ASCII码表 ……352
附录B 常用库函数 ……353
参考文献 ……356

# 第一部分 面向过程的程序设计

## 第1章 C++概述

### 1.1 计算机语言与程序

人类语言是人们之间交流信息的工具，而计算机语言是人与计算机交流信息的工具。用计算机解决问题时，人们必须首先将解决该问题的方法和步骤按一定序列和规则用计算机语言描述出来，形成计算机程序，然后计算机就可自动执行程序，完成所需要的功能了。

计算机语言与程序经历了以下三个阶段的发展。

#### 1.1.1 机器语言与程序

我们知道，在计算机内部采用二进制表示信息。指挥计算机完成一个基本操作的指令也是由二进制代码构成的，称之为机器指令。每一条机器指令的格式和含义都是设计者规定的，并按照这个规定设计制造硬件。一个计算机系统全部机器指令的总和，称之为指令系统，它就是机器语言。用机器语言编制的程序为如下形式:

```
0000    0100    0001    0010
0000    0100    1100    1010
0001    0010    1111    0000
1000    1010    0110    0001
.
.
.
```

每一行都是一条机器指令，代表一个具体的操作。用机器语言编写的程序，能直接在计算机上运行，运行速度快、效率高。但必须由专业人员编制程序。使用机器语言编写的程序紧密依赖硬件，程序的可移植性差。所谓移植，是指在一种计算机系统下编写的程序，经过改动可以在另一种计算机系统中运行，并且运行结果一样。改动越少，可移植性越好；改动越多，可移植性越差。

机器语言是第一代计算机语言。

#### 1.1.2 汇编语言与程序

汇编语言是一种符号语言，它将难以记忆和辨认的二进制指令代码用有意义的英文单词缩写来代替，英文单词缩写被称为助记符，每一个助记符代表一条机器指令。例如，用ADD表示加法，用SUB表示减法。用汇编语言编写的程序有如下形式:

```
MOV   AL   12D        // 表示将十进制数 12 送往累加器
SUB   AL   18D        // 表示从累加器中减去十进制数 18
.
.
.
HLT                   // 表示停止执行程序
```

汇编语言改善了程序的可读性和可记忆性，使编程者在编写程序时稍微轻松了一点。但汇编语言程

序不能在计算机上直接运行，必须把它翻译成相应的机器语言程序才能运行。将汇编语言程序翻译成机器语言程序的过程叫做汇编，汇编过程是计算机运行汇编程序自动完成的，见图1-1。

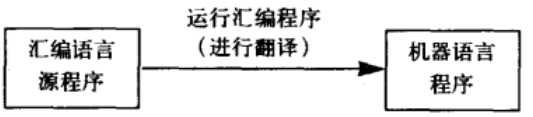



图1-1　汇编过程

汇编语言是第二代计算机语言。

### 1.1.3　高级语言与程序

如上所述，机器语言和汇编语言都是面向机器的语言，受机型限制，通用性差，学习也不容易，一般只适用于专业人员。为了从根本上解决这个问题，人们创造了高级程序设计语言，简称高级语言。高级语言用比较类似于人类自然语言和数学语言的方式描述问题、编写程序。例如，用C++语言编写的程序片段如下：

```
int a, b , c ;      // 定义变量 a, b, c
cin >> a >> b ;     // 输入变量 a, b 的值
c = a + b ;         // 将变量a, b 的值相加，结果赋给变量c
cout << c ;         // 输出求和结果
```

该程序片段每条语句的功能已给出说明。用高级语言编写程序时，不需要了解计算机的内部结构，只要告诉计算机“做什么”即可。至于计算机用什么指令去完成（即“怎么做”），编程者不需要关心。编写高级语言程序时，不需要考虑具体的计算机硬件系统。

但是，计算机无法直接执行高级语言程序，必须将高级语言程序翻译成机器语言程序才能执行。翻译过程分成两步即编译和连接，如图1-2所示。

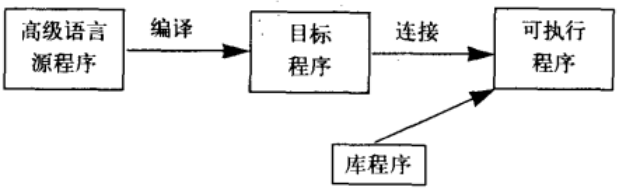



图1-2　编译、连接过程

高级语言源程序经编译后得到目标程序（一种形式的机器语言程序），再经过与库程序（包含通用函数）连接生成可执行的机器语言程序。

高级语言是第三代计算机语言。

高级语言不仅易学易用，通用性强，而且具有良好的可移植性。因为不同的计算机系统有不同的编译程序（“编译程序”也称为“编译器”），将高级语言源程序重新编译（在编译之前有时需对源程序稍加改动，称为移植）后，即可在不同的计算机系统中运行。

目前世界上有数百种高级语言，应用于不同领域，而C++作为其中的优秀语言得到了普遍地使用。

## 1.2　从C到C++

C语言是美国贝尔实验室的Dennis Ritchie在B语言的基础上开发出来的，1972年在一台DEC PDP-11计算机上实现了最初的C语言。当时设计C语言是为了编写UNIX操作系统，UNIX操作系统90%的代码由C语言编写，10%的代码由汇编语言编写。随着UNIX操作系统的广泛使用，C语言也被人们认识和接受。

C语言在各种计算机上的快速推广导致了许多C语言版本。这些版本虽然是类似的，但通常并不兼容。显然人们需要一个与开发平台和机器无关的标准的C语言版本。1989年美国国家标准协会（American National Standard Institute，ANSI）制定了C语言的标准（ANSI C）。Brian Kernighan和Dennis Ritchie编著的《The C Programming Language》（1988版）介绍了ANSI C的全部内容。

C语言具有如下特点：

（1）语言简洁、紧凑，使用方便、灵活。C语言只有32个关键字，程序书写形式自由。

（2）具有丰富的运算符和数据类型。

（3）C语言可以直接访问内存地址，能进行位操作，使其能够胜任开发操作系统的工作。

（4）生成的目标代码质量高，程序运行效率高。

（5）可移植性好。

但是，C语言也具有它的局限性：

（1）C语言数据类型检查机制较弱，这使得程序中的一些错误不能在编译时被发现。

（2）C语言本身几乎没有支持代码重用的语言结构，因此一个程序员精心设计的程序，很难为其他程序所用。

（3）当程序达到一定规模时，程序员很难控制程序的复杂性。

为了满足管理程序的复杂性需要，1980年贝尔实验室的Bjarne Stroustrup博士及其同事对C语言进行了改进和扩充。最初的成果称为“带类的C”，而后称为“新C”。1983年由Rick Mascitti提议正式命名为C++（C Plus Plus）。因为在C语言中，运算符“++”是对变量进行增值运算，那么C++的喻义是对C语言进行“增值”。1994年制定了ANSI C++ 草案。以后又经过不断完善，成为目前的C++，C++仍然在不断的发展中。

## 1.3 程序设计方法

C++语言包括过程性语言部分和类部分。过程性语言部分和C语言没有本质差别。类部分是C中没有的，它是面向对象程序设计的主体。要学好面向对象程序设计，首先必须具有过程性语言的基础。所以学习C++，首先必须学习其过程性语言部分，然后再学习类部分。

程序设计方法正在从结构化方法走向面向对象方法。C能够很好地支持结构化程序设计，而C++支持面向对象程序设计。

### 1.3.1 结构化程序设计方法

结构化程序设计的主要思想是：将任务按功能分解并逐步求精，分解成较小的、功能完整的模块，每一个模块用一个过程或函数完成。程序员把数据与程序分开存储，将程序看成处理数据的一系列过程。过程或函数由一条一条可顺序执行的指令组成。编写程序的主要技巧在于追踪哪些函数调用了哪些函数，哪些数据发生了变化。结构化程序设计方法能够较好地解决一些复杂问题。其主要缺点是，程序依赖于数据，当数据结构发生变化时，必须对程序进行修改。另外，当开发一个新任务时，适用于老任务的程序一般不能重复利用，从编程的角度来说需要重复投入，重新开发程序。而基于可重用指导思想的面向对象程序设计方法能够较好地解决这一问题。

### 1.3.2 面向对象的程序设计方法

面向对象的程序设计（Object Oriented Programming，OOP）方法是近年来十分流行的一种程序设计方法，它试图用客观世界中描述事物的方法来描述一个程序要解决的问题。对象是客观世界中一个实际存在的事物，比如一个具体的人就是一个对象，将一般人的共同属性抽象出来就可以构成“类”，如“人”类，它的静态属性有姓名、年龄、性别、身高、体重等，动态属性有思考、走路、说话、吃饭、睡觉等，将静态属性作为类的数据成员，而将动态属性作为类的执行代码。类是一个抽象的概念，而对象是类的具体实例，比如“人”类的一个对象就是指一个具体的人。

下面介绍面向对象的程序设计的要素。

#### 1. 封装性

将描述对象的数据及对这些数据进行处理的程序代码有机地组成一个整体，对数据及代码的存取权限加以限制，这种特性称为封装。封装可以使对象内部的数据被隐藏起来，只能通过对象的执行代码接口来处理对象的数据。这样可以使编程者只关心该对象的使用，而不必去关心其内部的数据及代码细节。

#### 2. 继承性

继承是面向对象技术能够提高软件开发效率的重要原因之一。通过继承已有类的特性可以产生一个新类，新类（或称子类、派生类）是建立在已有类（或称父类、基类）的基础之上的，继承了父类的数据及代码后，在父类的基础之上只要增加少量数据和代码就可以产生一个新类。基类的数据及代码在派

生类中可以重复使用。如果将开发好的类作为构件放到构件库中，在开发新系统时便可以直接使用或继承使用了，这称为代码重用。

### 3. 重载性

有两种形式的重载。一种是函数重载，指同名函数可以完成不同功能，如两个同名函数abs( )，分别可以用来求整数和实数的绝对值。另一种是运算符的重载，如加(+)号即可以实现整数、实数的相加，又可以实现复数、字符串的相加等。重载性使编程者的思维更加自然，可避免或减少记忆烦琐的函数名和运算符。

### 4. 多态性

多态性是指不同的对象在接收到相同的消息后，以不同的行为去应对。这里“接收到相同的消息”是指调用名称相同的函数，“不同的行为”是指函数实现的功能不同。

有两种形式的多态，第一种为静态多态，在上述重载性中指出的函数重载和运算符重载属于静态多态。

而第二种多态，即动态多态是指在基类中定义的代码，在被派生类继承后，可以表现出不同的行为。

例如，基类是“几何图形”，它具有“绘图”行为，但这个行为并不具有具体含义，因为并不知道具体画一个什么图形。从“几何图形”类派生出“椭圆”类或“矩形”类，在派生类中，自动具有“绘图”功能，然后在派生类中重新定义“绘图”功能，实现具体的绘制椭圆或矩形。于是在基类中的共同行为“绘图”，在派生类中具有不同的行为。

## 1.4　简单的C++程序介绍

下面通过一个简单的C++程序来说明C++程序的基本结构。

**例1.1**　一个简单的C++程序。

```
/* ------------------------------------------------------------------
   Li0101.cpp  该程序用于求一个数的平方
   ------------------------------------------------------------------
*/
#include  <iostream.h>
void  main( void )
{
   int  num, square ;          // 定义变量 num、square
   cout << "num=" ;            // 输出提示信息
   cin >> num ;                // 输入一个数，赋给变量 num
   square = num * num ;        // 求 num 的平方，结果赋给变量 square
   cout << " num 的平方为:" << square << '\n' ; // 输出num 的平方
}
```

上述简单C++程序由注释语句、编译预处理命令和主函数构成。

### 1. 注释语句

注释是对编程意图、程序功能、算法、语句的作用等所做的说明。注释有两种形式，一种是以两个斜杠“//”开头直到该行结束，在 // 和行末之间加注释，此种形式的注释只能在一行中书写；另一种是在“/*”和“*/”之间加注释，此种形式的注释可以跨多行书写。

### 2. 编译预处理命令

在本程序中“#include  <iostream.h>”表示文件包含，即编译时将系统头文件iostream.h 插入到本源程序头部。一般地，在程序中如果要用到系统预先定义的标准函数、符号或对象，在程序的头部均要包含相应的头文件。在本程序中包含头文件iostream.h，是因为在函数中要用到系统预先定义的、与数据的输入/输出有关的输入/输出流对象 cin和cout。

### 3. 主函数main( )

任一C++程序均有而且只有一个主函数，表示程序从该函数开始执行。main前面的void表示该main ( )函数没有返回值。函数体用花括号{ }括起来。在函数体中，按照算法写出语句，完成功能。

执行上述程序时，首先在屏幕上显示:

```
num=
```

等待用户输入一个整数，假如输入的是: 8 <Enter>（<Enter> 表示 <回车>），则显示:

```
num 的平方为: 64
```

**例1.2** 一个由两个函数构成的C++程序，源程序名为Li0102.cpp。

```
#include  <iostream.h>
int  sum(int x, int y)          // A
{
   int z;
   z=x+y;
   return z;                    // B
}
void  main( void )
{
   int  a, b, c ;               // 定义变量a和b
   a=3; b=5 ;                   // 给变量a和b赋值
   c = sum(a, b) ;              // C  求a与b之和，结果赋给变量c
   cout << c << '\n' ;          // 输出c
}
```

本程序由两个函数组成。程序从主函数main( )开始，当执行到C行时，发生函数调用，将实际参数a和b的值分别赋给形式参数x和y；流程转入A行执行函数sum( )，函数sum执行结束到达B行，将计算结果z返回主函数；程序流程返回到主函数中的C行，并将计算结果赋值给变量c，程序继续执行，执行的结果是输出8。

从本例看出一个C++程序由若干函数构成，程序的执行总是从main( )开始，当发生函数的调用及返回时，程序的执行流程在函数间跳转。

## 1.5 程序开发的步骤

目前，大多数的程序设计语言都提供了集成开发环境，编程者首先在集成开发环境中输入源程序，如上例源程序名为Li0102.cpp，一般C++源程序的扩展名为 .cpp。在集成环境中启动编译程序将源程序转化成目标程序，一般名为Li0102.obj，即主文件名与源程序主文件名一致，扩展文件名为 .obj。然后启动连接程序将目标程序与库程序（一般扩展名为 .lib）连接生成可执行程序，一般扩展名为 .exe。最后运行可执行程序。

在程序的开发过程的各个阶段，如编译、连接、执行，均有可能出现错误，当出现错误后，必须回到程序的编辑状态对源程序进行修改。

程序开发的步骤如图1-3所示。

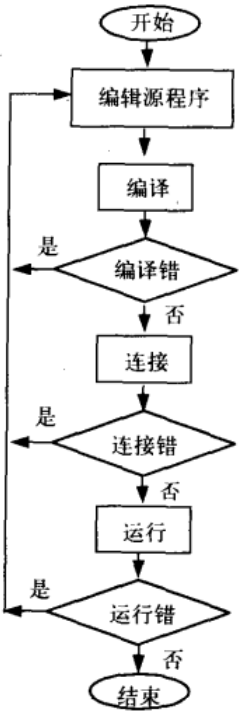



图1-3 开发C++程序的步骤

## 习题1

1. 程序设计语言经过了几代的发展？各代程序的书写形式有何变化？如何运行各代程序？

2. 什么是程序的可移植性？如何评价程序可移植性的优劣？
3. C语言的特点和局限性是什么？
4. 什么是结构化程序设计方法？
5. 什么是面向对象的程序设计方法？面向对象的程序设计的要素是什么？
6. C++程序的基本结构是什么？在程序中的两种注释形式如何？分别适用于什么场合？
7. 简要说明C++程序的开发步骤。

# 第2章 数据类型、运算符和表达式

学习编写程序，首先要了解构成程序的基本组成要素。从形式上说，C++程序是由一些符号、单词构成的语句组成；从逻辑上说，程序 = 数据+算法。所以必须首先了解构成程序的符号、单词、数据，本章主要介绍这些内容。

## 2.1 保留字和标识符

### 2.1.1 保留字

保留字（reserved word）也称为关键字（keyword），它们是C++预先定义的字符序列，具有特殊的含义及用法，编程者不能将它们用做自己的变量或函数名等。如例1.1中的类型说明符 int，用于定义整型变量。ANSI C++ 中共有48个保留字，各版本的C++有一些扩充。下面列出C++与Virsual C++ 兼容的43个保留字：

```
auto        break       case        char        class       const       continue
default     delete      do          double      else        enum        extern
float       for         friend      goto        if          inline      int
long        new         operator    private     protected   public      register
return      short       signed      sizeof      static      struct      switch
this        typedef     union       unsigned    virtual     void        volatile
while
```

这些保留字的意义和用法，我们后面将逐步介绍。

### 2.1.2 标识符

标识符（identifier）是有效字符序列，用来标识用户自己定义的变量名、符号常量名、函数名、数组名、类型名等。例如：在例1.2中，变量名a、b和c以及函数名sum均为用户定义的标识符。

标识符的命名应遵循以下规则：

（1）不能是保留字。

（2）只能由字母、数字和下划线三种字符组成。

（3）第一个字符必须为字母或下划线。

（4）中间不能有空格。

（5）最大长度为247个字符。一般不要太长，以不超过31个字符为宜。

（6）一般不要与C++中的库函数名、类名和对象名相同。

以下是合法标识符：

```
MyName          Average         GetDay
StudentName     _above          Lotus_1_2_3
```

以下是非法标识符：

```
M.D.John      $123       3D64      a>b      3DMax
```

通常为了增强程序的可读性，采用Windows标准命名法，因发明者的国籍为匈牙利，因此也称为匈牙利命名法。其规则是：标识符由多个英文单词组成，每个单词的第一个字母大写，其余为小写，如StudentName。

## 2.2 C++的基本数据类型

程序中的语句是处理数据的。描述一个数据需要两方面的信息：一是数据占用的存储空间的大小

（即该数据占用的字节数），二是该数据允许执行的操作或运算。为数据赋予类型就可以区分这两方面的信息。C++的数据类型分为两类，一类是基本数据类型，另一类是导出数据类型。基本数据类型包括字符型、整型、实型、双精度型。导出数据类型是由基本数据类型构造出来的数据类型，包括数组、指针、结构体和类等。表2-1中列出了VC++（即Visual C++）基本数据类型。

**表2-1　VC++中的基本数据类型**

| 类型标识 | 名　称 | 占用字节数 | 取值范围 |
|---|---|---|---|
| char | 字符型 | 1 | $-128 \sim 127$ |
| int | 整型 | 4 | $-2^{31} \sim (2^{31}-1)$ |
| float | 实型 | 4 | $-10^{38} \sim 10^{38}$ |
| double | 双精度实型 | 8 | $-10^{308} \sim 10^{308}$ |

字符型用来存放一个字符的ASCII码值，此时它代表一个字符；或者将其看成一个具有8位二进制码的整型变量。整型用来存放一个整数，其占用的位数随不同的C++版本不同而不同，可以占用2个字节或4个字节，在VC++中占用4个字节。实型用来存放实型数据，两种类型的实型数据因占用的字节数不同，其表示的数据范围也不同。

对于字符型和整型，我们均可以把它们看成整型数。在类型标识符char和int之前加上修饰词后，可以得到其他类型的整型数。这些修饰词有signed（有符号的）、unsigned（无符号的）、long（长的）、short（短的），组合后的数据类型见表2-2。

**表2-2　VC++中的全部基本数据类型**

| 类型标识 | 名　称 | 占用字节数 | 取值范围 | 有效数字 |
|---|---|---|---|---|
| [signed] char | 有符号字符型 | 1 | $-128 \sim 127$ | |
| unsigned char | 无符号字符型 | 1 | $0 \sim 255$ | |
| [signed] short [int] | 有符号短整型 | 2 | $-32\,768 \sim 32\,767$ | |
| unsigned short [int] | 无符号短整型 | 2 | $0 \sim 65\,536$ | |
| [signed] int | 有符号整型 | 4 | $-2^{31} \sim (2^{31}-1)$ | |
| unsigned [int] | 无符号整型 | 4 | $0 \sim (2^{32}-1)$ | |
| [signed] long [int] | 有符号长整型 | 4 | $-2^{31} \sim (2^{31}-1)$ | |
| unsigned long [int] | 无符号长整型 | 4 | $0 \sim (2^{32}-1)$ | |
| float | 实型 | 4 | $-10^{38} \sim 10^{38}$ | 7位 |
| double | 双精度实型 | 8 | $-10^{308} \sim 10^{308}$ | 15~16位 |

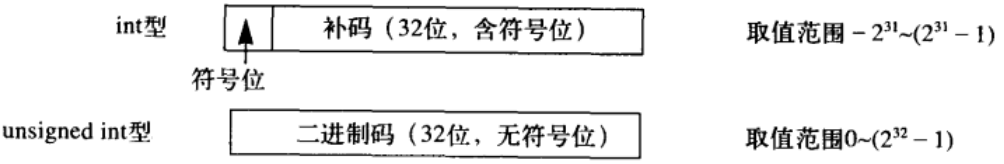
表2-2中各数据类型方括号中的内容可省略。

int型量是32位的，在内存中以补码方式存放。例如，5和－5作为整型量，在内存中存放的逻辑形式为：

```
5:     0000 0000 0000 0000 0000 0000 0000 0101
-5:    1111 1111 1111 1111 1111 1111 1111 1011
```

注意：补码的最高位是符号位。5是正数，其补码形式与原码相同。－5的补码是其原码取反加1。

注意：*int* 型量与 unsigned int 型量的区别。int型量是32位的，采用补码方式表示，有符号位；而unsigned int 型量也是32位的，但采用原码方式表示，没有符号位。




内存中有编码 1111 1111 1111 1111 1111 1111 1111 1011，如果把它赋给一个int整型量，它所代表的

值为－5；如果把它赋给一个unsigned int整型量，它所代表的值为4 294 967 291（等于$2^{32}-1-4$）。

## 2.3 常量和变量

### 2.3.1 常量

在程序的运行过程中，其值不变的量称为常量。编程者可以直接在程序中书写常量。如在例1.2中有常量3和5。下面将对各种类型的常量及其书写形式做详细说明。

**1. 整型常量**

（1）十进制整数，如：123，－456。

（2）八进制整数，如：0123，－016。

八进制整数以0（零）开头，在数值中可以出现数字符号0~7。

（3）十六进制整数，如：0x123，－0xAB。

十六进制整数以0x（零x）或0X开头，在数值中可以出现数字符号0~9、A~F（或小写的a~f）。

（4）长整型与无符号型整数。

长整型整数，如：12L，　0234L，－0xABL，12l，　0234l，－0xABl

无符号型整数，如：12U，　0234U，0xABU，　12u，　0234u，0xABu

在一个常数后加L或l（小写的L）表示该常数是长整型的整数；在一个常数后加U或u（小写的U）表示该常数是无符号型的整数。

**2. 实型常量**

实型常量在内存中以浮点形式存放，均为十进制数，无数制区分。有两种书写形式：

（1）小数形式：必须写出小数点，如1.65，1.，.123均是合法的实型常量。

（2）指数形式：也称为科学表示法形式，如$1.23\times10^5$和$1.23\times10^{-5}$在程序中可以表示成1.23e5或1.23e－5。e或E前必须有数字，e或E后必须是整型数；1000应写为1e3，而不能写成e3。

**3. 字符常量**

用单引号括起来的一个字符称为字符常量，如在程序中书写'a'、'A'、'?'、'#'分别表示4个字符。在内存中对应存放该4个字符的ASCII码值，其数据类型为char型。用这种方式只能表示键盘上的可输入字符，ASCII码表（见附录A）中的控制字符就不能用这种方式表示了，如ASCII码值为10的字符为控制字符，表示换行。为此，C++提供了另外一种表示字符常量的方法，即“转义”字符。转义字符是以反斜杠“\”引导的特殊的字符常量表示形式。一般地，'n'表示字母n，而'\n'仅代表一个字符，即表示一个控制字符“换行”，跟随在“\”后的字母n的意义发生了转变，所以叫做转义字符。在表2-3中列出了C++中预定义的转义字符。

**表2-3　C++中预定义的转义字符**

| 转义字符 | 名　称 | 功能或用途 |
|---|---|---|
| \a | 响铃 | 用于输出响铃 |
| \b | 退格（Backspace键） | 用于退回一个字符 |
| \f | 换页 | 用于输出 |
| \n | 换行符 | 用于输出 |
| \r | 回车符 | 用于输出 |
| \t | 水平制表符（Tab键） | 用于输出 |
| \v | 纵向制表符 | 用于制表 |
| \\ | 反斜杠字符 | 用于表示一个反斜杠字符 |
| \' | 单引号 | 用于表示一个单引号字符 |
| \" | 双引号 | 用于表示一个双引号字符 |
| \ddd | ddd是ASCII码的八进制值，最多三位 | 用于表示该ASCII码代表的字符 |
| \xhh或\Xhh | hh是ASCII码的十六进制值，最多两位 | 用于表示该ASCII码代表的字符 |

表2-3中最后4行是转义字符的高级形式，它可以表示任一字符。如 '\n' 表示控制字符“换行”，它的ASCII码是十进制数10，因此 '\n' 也可以表示成 '\12'、'\012'、'\xa'、'\0xa'。字母A的ASCII码是十进制数65，它的八进制数和十六进制数分别是101和41，所以在程序中，字母A可以写成 'A'、'\101'或'\x41'。

**4. 字符串常量**

字符串常量是用双引号括起来的字符序列，如: "CHINA"、 "How do you do."、"a"。字符串常量在内存中的存放形式是，存放其连续字符的ASCII码值，末尾加一个结尾标志 '\0'。'\0' 表示ASCII码值为0的字符，即ASCII码表中第一个字符，也称为“空字符”，其值为0。字符串"CHINA" 在内存中的存放形式如图2-1所示。

注意: 字符串常量 "a" 和字符常量 'a' 是不同的。字符常量 'a' 在内存中占有一个字节，而字符串常量 "a" 在内存中占有二个字节，如图2-2所示。

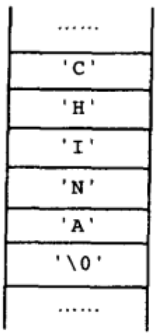



图2-1　字符串在内存中的存放

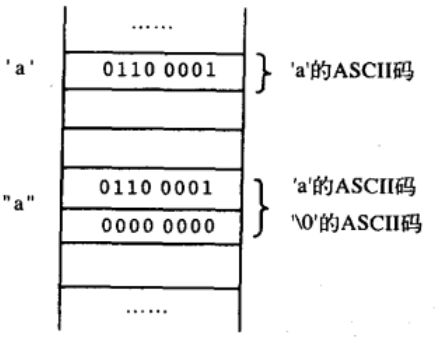



图2-2　字符和字符串在内存中存放方式的比较

## 2.3.2　符号常量

我们可以在程序中直接书写常量，但有时会遇到一些麻烦。如在做数学计算时，在程序中要多次使用π，需要多次输入3.1415926，这样输入程序时可能出现将数字输错的情况，同时如果精度上需要变化，如将3.1415926改为3.14，就需要在程序中进行多处修改。

于是C++提供了这样一种机制，避免上述麻烦。用一个标识符代表一个常量，称为符号常量。编程者可以在程序的开头定义一个符号常量，令其代表一个数值，在程序的后面使用该符号常量。当然，符号常量一旦定义并赋初值以后，就不可以改变其值了。

在C++中有两种形式定义符号常量: 一种是使用编译预处理命令；另一种是使用C++中的常量说明符const。例如:

```
#define   PRICE   30
#define   PI       3.1415926
#define   S        "China"
const  f loat  pi = 3.14 ;
```

此4行定义了4个常量PRICE、PI、S和pi。给常量取一个有意义的名字是为了提高程序的可读性。另外如果在程序中多处使用了同一个常量，当需要对该常量进行修改时，只需在定义处修改一次，而不需要修改多处。下面用一个例子说明符号常量的使用。

**例2.1**　符号常量的使用。

```
#include <iostream.h>
#define  PI  3.1415926
void main(void)
{
```

```
    double  angle, radian;
    cin >> angle;                  // 输入角度
    radian = angle * PI / 180;     // 将角度转化成弧度
    cout << radian << '\n';        // 输出弧度
}
```

### 2.3.3 变量

在程序的运行过程中，其值可变的量称为变量。变量名必须用标识符来标识。变量根据其取值范围的不同可分为不同类型的变量：如字符型变量、整型变量、实型变量等。不同类型的变量其存储空间是不同的。所以一个变量有三个要素，即变量名、它的存储空间、在存储空间中存放的该变量的值。

#### 1. 定义变量

定义变量的一般格式为：

```
[<存储类别>] <变量类型> <变量名1>,<变量名2>, ..., <变量名n >
```

我们暂时不必关心变量的<存储类别>，在后面有关章节中会有详细说明。现在我们定义变量如下：

```
int  a, b ;                // 定义2个整型变量a, b
unsigned  u ;              // 定义1个无符号整型变量u
float  f ;                 // 定义1个单精度实型变量f
double  d ;                // 定义1个双精度实型变量d
char  c1, c2, c3 ;         // 定义3个字符型变量c1, c2, c3
```

变量必须先定义后使用，原因是：

(1) 变量定义后它就有类型了，这样编译器才可以根据它的类型来给它分配存储空间，并建立变量名和其存储空间的对应关系，这样就可以给变量赋值或读取该变量的值了。

(2) C++中某些运算对参加运算的数据类型有限制，变量有类型后，便于编译器对运算的合法性做检查。

#### 2. 变量赋初值

当使用变量时，变量必须有值，给变量赋初值的方法有两种：

(1) 变量定义后，用赋值语句赋初值。例如：

```
int  a, b ;
a = 12; b = -24 ;
char c1,c2 ;
c1='A'; c2='B' ;
```

此处“=”是赋值运算符，表示将等号右边的值放入变量a的存储空间中。

(2) 在定义变量的同时，直接赋初值（称为变量的初始化）。例如：

```
int  a=12, b= -24 ;
char c1='A', c2='B' ;
```

## 2.4 基本运算符和表达式

### 2.4.1 C++运算符及表达式简介

在C++中，数据处理是通过运算来实现的。运算符又称为操作符。为表示一个计算过程，需要使用表达式，表达式是由操作符、操作数构成的一个计算序列。在C++中有很多运算符，如算术运算符（+ - * / %）、关系运算符（> >= < <= == !=）等。在表2-4中列出了C++中各种运算符及其优先级和结合性。

表2-4　C++的运算符及其优先级和结合性

| 优先级 | 运算符 | 结合性 |
|---|---|---|
| 1 | ( ) . -> [ ] :: .* ->* &（引用） | 左→右 |
| 2 | *（间接访问） &（取地址） new delete ! ~ ++ -- - sizeof( ) | 右→左 |
| 3 | *（乘） /（除） %（求余）（算术运算符） | 左→右 |
| 4 | + -（算术运算符） | 左→右 |
| 5 | << >>（位运算符） | 左→右 |
| 6 | < <= > >=（关系运算符） | 左→右 |
| 7 | == !=（关系运算符） | 左→右 |
| 8 | &（位运算：与） | 左→右 |
| 9 | ^（位运算：异或） | 左→右 |
| 10 | \|（位运算：或） | 左→右 |
| 11 | &&（逻辑运算：与） | 左→右 |
| 12 | \|\|（逻辑运算：或） | 左→右 |
| 13 | ?:（条件运算，三目运算符） | 右→左 |
| 14 | = += -= *= /= %= <<= >>= &= ^= \|=<br>（赋值及复合赋值运算符） | 右→左 |
| 15 | ,（逗号运算符） | 左→右 |

若一个运算符只能对一个操作数进行，则称其为一元运算符或单目运算符，如表2-4中的“+”正号和“-”负号运算符。若一个运算符要求两个操作数，则称其为二元运算符或双目运算符，如表2-4中的“+”加号和“-”减号运算符。若一个运算符要求三个操作数，则称其为三元运算符或三目运算符，如表中的“?:”条件运算符。

### 2.4.2　算术运算符和算术表达式

C++提供了五个算术运算符，它们是：+（加）、-（减）、*（乘）、/（除）、%（求余），它们是二元运算符，其中+（正号）和-（负号）又可用做一元运算符。

注意：

对于除法运算，当两个运算量均为整数时为整除。例如 5/2 得到结果 2。当至少有一个运算量为实数时则为通常意义的除法。例如 5.0/2 得到结果 2.5。

对于求余运算（或称模运算），参加运算的量必须为整型数据。例如，8%3 结果为2。而8.0%3为非法表达式。

### 2.4.3　运算优先级和结合性

C++表达式中可以出现多个运算符和运算量，计算表达式时必须按照一定的次序，运算符的优先级和结合性规定了运算次序。

所谓优先级是指，在同一个表达式中出现了不同级别的运算符，先算哪一个？一般规定首先计算优先级较高的。

如：d = a + b * c 表达式中出现了三个运算符即 *（乘）、+（减）、=（赋值），这三个运算符的优先级由高到低依次是 *、+、= ，所以先算乘，再算加，最后执行赋值运算，即将赋值运算符右边的表达式的值赋给变量 d 。

注意：括号可以改变运算符的优先级。如：d = (a + b) * c表达式的运算次序是+、*、=。

所谓结合性是指，在表达式中若连续出现若干个优先级相同的运算符，应该先做那一个的规定。

如：d = a + b - c 表达式中出现了三个运算符，即 +（加）、-（减）、=（赋值），其中加和减的运算优先级相同，而赋值运算符优先级较低。从表2-4中看出，加和减运算的结合性是从左向右的，因此计算该表达式时，先算加，再算减，最后进行赋值。

### 2.4.4 关系运算符和关系表达式

“关系运算符”实际上就是“比较运算符”。关系运算符的意义及其运算优先次序如下：

< 小于
<= 小于等于
> 大于
>= 大于等于
（运算优先级较高）

== 恒等于
!= 不等于
（运算优先级较低）

关系表达式是用关系运算符连接两个表达式构成。如：a>3.0，a+b>b+c，(a=3)>(b=5)，'a'<'b' 等。

关系运算符的运算优先级比算术运算符的优先级低，但比赋值运算符的优先级高。

参加关系运算的两个操作数可以是任意类型的数据。当比较结果成立时，结果为 1（表示“真”）；当比较结果不成立时，结果为 0（表示“假”）。例如，若有int a=1, b=2, c=3;则表达式 a>b 的值为 0；表达式 b<a+c 的值为 1；表达式 a==b-1 的值为 1；表达式 a>b>c 的值为 0。

### 2.4.5 逻辑运算符和逻辑表达式

C++语言中提供了三种逻辑运算符，它们是：

! 逻辑“非”（一元运算符）
&& 逻辑“与”（二元运算符）
|| 逻辑“或”（二元运算符）

三种逻辑运算的意义是：若a和b是两个参加运算的量，a&&b 的意义是，当 a、b 均为真时，表达式的值为真；a || b 的意义是，当 a、b 均为假时，表达式的值为假；!a的意义是，当a为真时，!a为假；当a为假时，!a为真。可以用如表2-5所示的“真值表”来概括这几种含义。

表2-5 逻辑运算“真值表”

| a | b | a&&b | a \|\| b | !a |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

逻辑表达式是用逻辑运算符连接两个表达式构成。如：a>b&&x>y、a==b || x==y、!a >b，逻辑运算符的运算优先级比算术、关系运算符的优先级低，但比赋值运算符的优先级高；三个逻辑运算符的优先级由高到低依次是 !、&&、||。而取非“!”运算的优先级比算术、关系运算符还要高。因此，上述三个逻辑表达式的意义等价于(a>b)&&(x>y)、(a==b) || (x==y)、(!a) >b。

参加逻辑运算的两个操作数可以是任意类型的数据。进行逻辑运算时，在判断参加运算的量的逻辑真假性时，规定：任何非0值表示逻辑真，0值表示逻辑假。例如：a=-1; b=2.0;则a为真，b为真，从而表达式 a&&b 的结果也为真。

C++在给出关系表达式或逻辑表达式运算结果时，以数值 1 代表“真”，以数值 0 代表“假”。

注意：在C++程序中，要表示数学关系 $0 \leqslant x \leqslant 10$ 时，C++的逻辑表达式必须写成 0<=x && x<=10，而不能写成 0<= x <=10。因为C++的运算次序是有规定的，而不是想当然的。例如当 x = -1时，数学关系$0 \leqslant x \leqslant 10$显然是不成立的；而C++ 表达式0<= x <=10的运算结果为真，产生矛盾。但此时C++表达式 0<=x && x<=10结果为假，与数学关系保持一致。

### 2.4.6 位运算符和位运算表达式

位运算是对整型数据的运算，而且规定符号位参与运算，主要用于设计系统程序。位运算符共有6种，它们是：按位与（&）、按位或（|）、按位异或（^）、按位取反（~）、左移（<<）、右移（>>）。下面一一介绍。

#### 1. 按位与（&）

运算符“&”将其两边的操作数的对应位逐一按位进行逻辑与运算。每一位二进制数（包括符号位）

均参加运算。

例如：
```
char  a=3, b=-2 , c ;    // 此时，可将a、b、c看成是一个字节长度的整型数
c = a & b ;
a  0000 0011
& b  1111 1110
c  0000 0010
```

结果，变量c的值为2。

### 2. 按位或（|）

运算符“|”将其两边的操作数的对应位逐一按位进行逻辑或运算。每一位二进制数（包括符号位）均参加运算。

例如：
```
char  a=18, b=3 , c ;    // 此时，可将a、b、c看成是一个字节长度的整型数
c = a | b ;
a  0001 0010
| b  0000 0011
c  0001 0011
```

结果，变量c的值为19。

### 3. 按位异或（^）

运算符“^”将其两边的操作数的对应位逐一按位进行逻辑异或运算。每一位二进制数（包括符号位）均参加运算。异或运算的定义是，若对应位相异，结果为1；若对应位相同，结果为0，

例如：
```
char  a=18, b=3 , c ;    // 此时，可将a、b、c看成是一个字节长度的整型数
c = a ^ b ;
a  0001 0010
^ b  0000 0011
c  0001 0001
```

结果，变量c的值为17。

### 4. 按位取反（~）

运算符“~”是一元运算符，结果将操作数的对应位逐一取反。

例如：
```
int a=18, b ;
b = ~ a ;
~ a  0000 0000 0000 0000 0000 0000 0001 0010
  b  1111 1111 1111 1111 1111 1111 1110 1101
```

结果，变量b 的值为－19，或记为十六进制数 0xffffffed。

### 5. 左移（<<）

设a、n是整型量，左移运算的一般格式为：a<<n，其意义是，将a按二进制位向左移动n位，移出的高n位舍弃，最低位补n个0。

例如，short int a=15,x;,a的二进制形式是0000 0000 0000 1111，做x = a<< 3;运算后x的值是0000 0000 0111 1000，其十进制数是120。左移一个二进制位，相当于乘以2操作。左移n个二进制位，相当于乘以$2^n$操作。程序运行时，左移n位比乘以$2^n$操作速度快。

左移运算有溢出问题，因为整数的最高位是符号位，当左移一位时，若符号位不变，则相当于乘以2操作，但当符号位变化时，就发生了溢出。例如，若有char a = 127, x ;，即a的二进制形式是 0111 1111，做x= a<<2; 运算后x的值1111 1100，表示的十进制数是－4，此时即发生了溢出。原因是，对8位二进制补码，其表示的数的范围是－128 ~ +127，a的值是127，若将a的值乘以$2^2$，则超出了所能表示的数的范围。溢出后，变量a所表示的数必须根据它在内存中的存储形式进行判断。

### 6. 右移（>>）

设a、n是整型量，右移运算的一般格式为：a>> n，其意义是，将a按二进制位向右移动n位，移出的低n位舍弃，高位补0还是1呢？这取决于a是什么类型的整型数，若a是有符号的整型数，则高位补符号位；若a是无符号的整型数，则高位补0。

例如：
```
char  a = -4 , b=4, x, y ;
x = a >> 2;
y = b >> 2;
a: 1111 1100   →  x: 1111 1111
b: 0000 0100   →  y: 0000 0001
```

x的值是－1，y的值是1。右移一位相当于除以2操作。

又例如：
```
unsigned char a = -4 , x ;
x = a >> 2;
a: 1111 1100   →  x: 0011 1111
```

x的值是63。此时，右移一位若符号位发生了变化，称为溢出，就不表示除以2操作了。

## 2.4.7 自增、自减运算符和表达式

C++中还有两个可以改变变量的值的运算符，它们是自增（++）和自减（－－）运算符，作用是使变量的值加1或减1。这两个运算符均是一元运算符，而且只能对变量做运算，可以放在变量之前或之后。如：++i、－－i表示先将i的值加1或减1，然后再参加运算；而i++、i－－表示先用i的值参加运算，然后再加1或减1。

例如：int i =3, j;j=++i;，则运算结束后，i的值是4，j的值也是4。

又如：int i =3, j;j=i++;，则运算结束后，i的值是4，j的值是3。

又如：int i =3, j=4;x=(i++)+(j++);，则运算结束后，i、j的值是4、5，而x的值是7。

注意，自增、自减运算符只能作用于变量，如果写出表达式 3++ 或 ++(x+y) 就是非法的，原因是在这两个表达式中，++作用在常量和表达式上了。

## 2.4.8 赋值运算符和赋值表达式

### 1. 赋值运算符

在C++中，“=”是赋值运算符，赋值表达式是由赋值运算符连接一个变量和一个表达式构成，其格式是：

<变量> <赋值运算符> <表达式>

如：a = 8 和a = b + c。

赋值表达式的求解过程是：求出 <表达式> 的值，赋给 <变量>。

赋值表达式的值是：<变量> 的值。

注意，赋值表达式中的<表达式>还可以是另外一个赋值表达式。赋值运算符“=”与数学上的等号意义不一样，赋值表示一种运算，如 i = i + 1，表示先计算等号右边的i+1，再把结果赋给变量i。赋值运算符的优先级比上述所有运算符的优先级都低，其结合性是从右向左的，如表达式：

```
b = c = d = a+5
```

它是一个合法表达式，若a的初值是3，则先计算表达式a+5，结果是8，将8赋给变量d，此时赋值表达式“d = a+5”的值是8，再从右向左依次计算c=8，b=8，结果整个表达式的值是8。又如表达式：

```
a = 5 + c = 6
```

它是非法表达式，因为按照运算符的优先级应先算5+c，再算表达式最右边的赋值“=”，即将6赋给“5+c”，但这是一个非法赋值。

例如: 表达式a=b=5 的值是5; 表达式a=5+(c=6) 的值是11; 表达式a=(b=4)+(c=6)的值是10。

**2. 复合赋值运算符**

表达式a = a + 3 可简写成 a + = 3，表达式a = a * b可简写成 a * = b。在C++中，所有的二元算术运算符和二元位运算符与赋值运算符都可以组合成一个复合的赋值运算符，它们是:

+=（加等）　-=（减等）　*=（乘等）　/=（除等）　%=（求余等）

<<=（左移等）　>>=（右移等）　&=（与等）　^=（异或等）　|=（或等）

“复合赋值运算符”与“赋值运算符”的优先级和结合性是一样的。如y *= x + 8，等价于y *= (x + 8)，也等价于y = y * (x + 8) 。又如表达式:

```
a += a-= a * a
```

如果a初值为2，则先计算a*a，值为4，再计算a-= 4，结果a的值变成-2，同时表达式a-= 4的值也是-2，再计算a += (-2)，结果a的值是-4，整个表达式的值也是-4。

### 2.4.9 逗号运算符和逗号表达式

逗号运算符: ,

逗号表达式格式: <表达式1> ,<表达式2>,...,<表达式n>

求解过程: 依次计算 <表达式1>，<表达式2>，...，<表达式n>的值。

逗号表达式的值: 为 <表达式n> 的值。

如逗号表达式: a=3*5, a*4,a+5 ，依次计算表达式a=3*5、表达式a*4、表达式a+5。

运算结束后: 变量a的值为15，整个表达式的值为20。

逗号运算符的优先级在C++中所有的运算符中是最低的，它的结合性是从左向右。

写出下面表达式在运算结束后a=?，x=?，表达式=?

```
a=3*5, a*4     是逗号表达式，运算结束后a=15，表达式的值为60
x=(a=3, 6*3)   是赋值表达式，运算结束后a=3,  x=18，表达式的值为18
x=a=3, 6*3     是逗号表达式，运算结束后a=3,  x=3，表达式的值为18
```

### 2.4.10 sizeof( ) 运算符和表达式

sizeof ( )运算符是一元运算符，它用于计算某类型操作数所占用的字节数。

格式: sizeof (<类型标识>/<变量名>)

功能: 求某类型操作数所占字节数。

例:
```
int  i ;
double  x;
```
sizeof(int) 和sizeof(i)均合法，结果均为4。

sizeof(double) 和sizeof(x)均合法，结果均为8。

### 2.4.11 逻辑表达式运算优化时的副作用

在C++中，在计算逻辑表达式的值时，是从左向右扫描表达式的，一旦能确定表达式的值后，就不继续进行计算了。

例: 求<表达式1> && <表达式2> 的值。

计算时，从左向右扫描，先计算<表达式1>，当<表达式1>为真时，继续计算<表达式2>; 当<表达式1>为假时，即能确定整个表达式的值为假，则停止计算< 表达式2 >。

又例: 求<表达式1> || <表达式2> 的值。

计算时，从左向右扫描，先计算<表达式1>，当<表达式1>为假时，继续计算<表达式2>; 当<表达式1>为真时，即能确定整个表达式的值为真，则停止计算< 表达式2 >。

例如:
```
int x , y , z , w ;
    x = y = z = 1 ;
```

```
w = ++x || ++y && ++z ;    //A
```

计算结束后，变量x、y、z和w的值分别是：2、1、1和1。因为在计算A行表达式时，先计算++x，结果是2，即值为真，其后紧跟或（||）运算符，但不继续往右计算了，y和z的值保持不变。等号右边逻辑表达式的值为“真”，此时变量w得值1。

## 2.5 类型转换

### 2.5.1 赋值时的自动类型转换

如果赋值运算符两侧的类型不一致，则遵循以下几条原则进行类型转换后赋值：

（1）实型数据赋给整型变量时，简单舍弃小数部分，将整数部分赋给整型变量，不进行四舍五入。如：`int i=3.96;`，则i 得值3。

（2）整型数据赋给实型变量时，数值不变，有效数字位数增加。

如：`float f=23;`，则f为23.0，有7位有效数字；如`double d=23;`，则d为23.0，有15~16位有效数字。

（3）整型数据之间相互赋值。

整型数据有[signed]char、unsigned char、[signed]short、unsigned short、[signed]int、unsigned int、[signed]long、unsigned long。各种类型的整型数占用的字节数是不同的，其二进制位数有长有短。它们之间相互赋值，其实就是它们内存数据之间的赋值，分两种情况：

- **“长的”整型量赋给“短的”整型量。**

所谓“长的”整型量是指该整型量的二进制位数较多，所谓“短的”整型量是指该整型量的二进制位数较少。将“长的”整型量赋给“短的”整型量时，方法是“低位截断”，将“长的”整型量的高位去掉，截取其与“短的”整型量相同位数的低位，然后进行赋值。

如：int整型量赋给字符型变量。

例：`char c = 250 ;`

250为int整型常量，在内存中的存储形式是32位二进制数：

0000 0000 0000 0000 0000 0000 1111 1010

可将变量c看成是8位有符号二进制整型量，赋值原则是取250内存表示形式的低8位赋给c，此时c中的值是1111 1010，我们知道C++中整型变量是以补码方式存放的，因此，变量c的值是 -6。

例：`short int a = 65536 ;`

就常数65 536本身来说，它是int整型量，因为它在int型量的数值表示范围内，它的值就是$2^{16}$，在内存中的形式是：0000 0000 0000 0001 0000 0000 0000 0000。而短整型变量a在内存中表示成16位的，所以截取65 536内存表示形式的低16位赋给a，此时a中的16位全为0，那么a的值是0，这就叫做赋值溢出。65 536超过了短整型量所表示的数的范围（-32 768 ~ 32 767）。

- **“短的”整型量赋给“长的”整型量。**

又分成两种情况：

① 将“短”的无符号整型量赋给“长”的整型变量（注意：此处的“长”和“短”的意义是：整型量所占的二进制位数），例如：

```
unsigned char c = -4 ;
int i ;
i = c ;
```

此例中涉及两种赋值，首先将“长”的整型量-4赋给“短”的变量c，c获取的值是-4内存表示形式的低8位；再将“短”的c变量的值赋给“长”的整型变量i。因为c是无符号整型变量，占8位，而i是有符号int型变量，占32位，此时在c的内存内容前补0使其扩展到32位后，赋值给变量i。赋值过程中各变量的内存表示形式如下：

```
-4: 1111 1111 1111 1111 1111 1111   1111 1100
 c:                                 1111 1100
 i: 0000 0000 0000 0000  0000 0000 1111 1100
```

此时，变量i的值是252。

② 将短的有符号整型量赋给长的整型变量。

此种情况只需符号位扩展，即在短的量前补符号位，使其长度达到长的整型量的长度，然后赋值。例如:

```
char c = -4 ;
int i ;
i = c ;
```

赋值过程各变量的内存内容如下:

```
-4: 1111 1111 1111 1111 1111 1111  1111 1100
 c:                                1111 1100
 i: 1111 1111 1111 1111 1111 1111 1111 1100 (扩展负号)
```

此时，变量i的值是-4。

又例如:

```
char c = 4 ;
int i ;
i = c ;
```

赋值过程各变量的内存内容如下:

```
 4: 0000 0000 0000 0000  0000 0000 0000 0100
 c:                                0000 0100
 i: 0000 0000 0000 0000  0000 0000 0000 0100 (扩展正号)
```

此时，变量i的值是4。

### 2.5.2 各种类型运算量混合运算时的自动类型转换

C++语言中有各种数据类型，它们的常量和变量之间可以混合运算。两个量运算时，在计算机内部首先将它们转换成相同数据类型的量，然后进行运算。虽然这种转换是C++内部自动完成的，但是若编程者知道了转换机理，对掌握及灵活运用C++表达式是有帮助的。转换原则如图2-3所示。

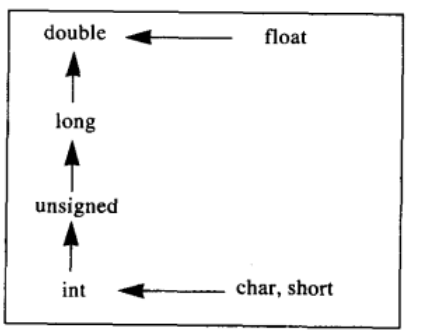


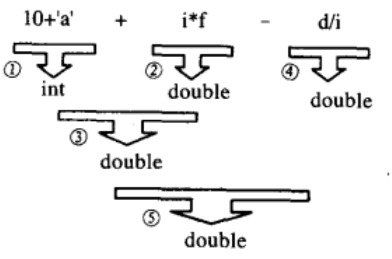

图2-3　混合运算类型转换原则

图2-3中横向向左的箭头表示必定转换。如:

```
char  c1, c2 ;
```

那么在做 c1+c2 运算时，首先将c1、c2的值均转换成int型，再将两个int型量相加。

图2-3中纵向箭头表示不同数据类型混合运算时的转换方向。原则是由低类型向高类型转换。所谓低类型是指，占用的存储字节少，所表示的数据范围小的类型。所谓高类型是指，占用的存储字节多，所表示的数据范围大的类型。例如: 如果有int i; float f;double d ;，则下述表达式的运算顺序和类型转换过程为（见右图):




先算第①步，结果是int型量，再依次计算②③④⑤步，最终整个表达式的结果类型是double型的。此时大家应该能理解5/2得2，5.0/2得2.5的道理了。

### 2.5.3 强制类型转换

前面介绍了不同类型量相互赋值时以及混合运算时的自动类型转换，但有时为了强调类型的概念或者为了满足运算符对数据类型的要求，可以将类型转换显式地写出。

格式是: <类型名> (<表达式>)

或 (<类型名>) <表达式>

例:
```
int i, a ; float x, y ; double  z ;
    i = (int) (x+y) ; 或i = int (x+y) ;
    z = (double) a ;  或z = double (a) ;
    a = (int) z% I ;  或a = int(z) % i ;
```

注意: 强制转换的对象是表达式的值，而不是改变变量本身的类型。

表达式double(a)的值为 double 型，而a仍然为int型变量。

## 习题2

1. 请找出以下合法的常量:

| | | | |
|---|---|---|---|
| 65535 | 3.5U | 66L | 1.24e－2 |
| 6e1.2 | '@' | 'abc' | "abc" |
| "?" | 892. | .123 | －0xAB |

2. 请找出以下合法的用户自定义的标识符:

| | | | |
|---|---|---|---|
| Max | _301 | 4_5 | M－1 |
| Char | int | my name | M.D.Jhon |

3. 若有 int x=12, y=77, z= －116，请用二进制表示这三个量的存储值。
4. 已知int a=2, b=3; float x=3.5, y=2.5; 请写出表达式 float(a+b)/2+int(x)%int(y) 的运算结果，并指出表达式运算结果的类型。
5. 已知float x=2.5, y=4.7; int a=7; 请写出表达式x+a%3*int(x+y)%2/4 的运算结果，并指出表达式运算结果的类型。
6. 已知int a=8, n=5，请写出下面表达式运算结束后，a、n以及表达式的值

| | | | |
|---|---|---|---|
| (1)a+=a | (2)a－=2 | (3)a*=2+3 | (4)a/=a+a |
| (5)a%=(n%=2) | (6)a+= a－=a*=a | (7) a=3*5, a*4 | (8) n=(a=3, 6*3) |
| (9)n=a=3, 6*3 | (10) a = ++a ‖ ++n | | |

7. 已知char ch=277 ; 请写出ch的内存内容的二进制形式。
8. 已知char ch=249 ; int i =ch; 请写出i的内存内容的二进制形式。
9. 已知unsigned char ch=249 ; int i =ch; 请写出i的内存内容的二进制形式。
10. 将下面的表达式看成逻辑表达式，已知int a=1, b=2, c=3; 请写出逻辑表达式的运算结果。

| | |
|---|---|
| (1) a+b>c&&b==c | (2) allb+c |
| (3) !(a>b)&&!cll－2 | (4) !(x=a)&&(y=b) |
| (5) 'a'+'5' | (6) !(a+b)+c－1&&b+c/2 |
| (7) a+b, a/2 | (8) a==(c－b) |

# 第3章　简单的输入／输出

通过前面的章节我们了解了组成一个程序的基本数据元素，下面我们就可以在数据元素的基础上开始学习编写简单的C++程序了。本章主要介绍计算机程序设计的基础：在C++程序中实现数据输入／输出的基本方法。由于C++中数据输入／输出的实现机制与后续章节中的面向对象重载等技术有着密切的联系，因此在没有系统地学习面向对象设计思想之前无法对此进行详细的介绍。因此，本章只对C++中数据输入／输出的使用方法做简单的介绍。有关C++编译器提供的完整的输入／输出流体系结构的实现原理，将在第14章中进行详细的介绍。

## 3.1　传统的输入／输出函数实现方法

由于计算机是一种信息处理机器，它的主要工作就是处理各种信息和数据。而这些信息不可能都是由计算机自己生成的，因此几乎每个实际应用的程序都需要有将用户的信息输送给计算机的功能。同时，计算机的最终运算结果也不可能只是供计算机自己使用，它必须以某种方式显示出来（显示在屏幕上或者打印到纸上），以供用户察看。因此，在所有实际应用的程序中，最基本的功能就是输入和输出功能。一个实际应用的程序应该具有接受用户输入数据的能力（包含零个或者多个输入），运算的结果必须要以某种方式显示给人看（至少包含一个或者包含多个输出）。一般来说，不包含输入／输出功能的程序，是一个没有实际使用价值的程序。

程序在执行期间，接收外部信息的操作称为程序的输入（Input），而把程序的数据向外部发送的操作称为程序的输出（Output）。由于输入和输出功能往往是成对出现的，因此我们经常会用英文缩写组合来表示它们——I/O。

由于C++是在C语言的基础上发展扩充而来的，为了保证程序的兼容性，它也支持C语言中的输入／输出的实现方法。在C语言中没有定义输入／输出语句，输入／输出功能是通过输入／输出系统库中的scanf( )和printf( )等函数来实现的，其中scanf( )函数负责输入操作，printf( )函数负责输出操作。我们称使用C语言库函数实现程序输入／输出功能的方法为传统的输入／输出实现方法。

在C语言中使用系统库函数，一般要包含相应的头文件，但是对于scanf( )和printf( )函数例外。这两个函数可以在不包含任何头文件的情况下使用。但是，C++编译器对程序的语法检查要远远比C语言编译器来得严格，在C++编译器中编译包含scanf( )和printf( )函数而又没有包含对应的头文件的程序时，往往编译器会报告错误，这一点学习过C语言程序设计的程序员必须特别注意。在C++编译器中使用任何一个系统库函数，都需要包含相应的头文件。程序例3.1中举例说明了传统的输入／输出函数在应用程序中的基本使用方法。

**例3.1**　演示传统的输入／输出函数的使用方法。

```
#include <stdio.h>
void main( )
{
  char a;
  printf("Please input a character:");
  scanf("%c", &a);                    // "%c"表示输入的变量a是char型的
  printf("The character is: %c \n", a);
}
```

由于scanf( )和printf( )函数的使用方法比较复杂，因此，在C++中利用先进的重载（overloading）等

面向对象技术，重新设计了一套相对便于使用的输入／输出方法，称为输入／输出流技术。限于篇幅，本书不再介绍scanf( )和printf( )函数的用法，对此感兴趣的读者可参考C语言的相关书籍。本章重点介绍C++中特有的输入／输出流技术。

使用C++编程时，建议尽量使用C++提供的输入／输出流来实现输入／输出操作。若在一个程序中同时使用C++的输入／输出流技术和C语言的输入／输出函数来完成输入／输出操作，有时会出现一些异常现象，如输入／输出语句的执行顺序不对等。当学习Visual C++高级编程时，读者也会知道如果不使用输入／输出流等C++特有的技术，将无法在程序中利用Visual C++提供的很多有用的跟踪调试等辅助工具。

同时，由于输入／输出流的书写格式简洁，阅读方便，符合现代软件设计中的可读性要求，因此再次建议读者在书写C++程序的时候彻底放弃C语言的输入／输出函数。

## 3.2 cin

计算机的输入／输出分为两大类：标准输入／输出和文件输入／输出。键盘和显示器分别是计算机的标准输入和标准输出设备，所以在键盘和显示器上的输入／输出称为标准的输入／输出。数据在磁盘上是以文件为单位存放的，所以磁盘数据的输入／输出称为文件的输入／输出。由于在一个系统中，键盘和显示器一般都只有一个，从键盘获取数据，将结果显示到显示器上，因此操作起来比较简单。而在磁盘访问中，往往需要同时对多个文件进行操作，所以操作起来比较复杂。本章是C++学习的基础，所以只讨论简单的标准输入／输出的操作方法。

C++中没有专门的输入／输出语句，所有的输入／输出操作都是通过输入／输出流来实现的。在输入／输出流中，输入操作是通过流cin来实现的，而输出操作是通过流cout来实现的。cin是由c和in两个单词组成的，代表C++的输入流；cout是由c和out两个单词组成的，代表C++的输出流。

由于cin和cout都是在头文件iostream.h中定义的，前面我们提醒过，C++中任何系统提供的操作或者函数的使用都需要加入对应的头文件说明；因此，要使用C++提供的输入／输出流技术时，必须在程序的开头增加一行说明如下：

```
#include <iostream.h>
```

即在程序中首先包含输入／输出流的头文件iostream.h。关于包含文件的作用，在第6章将进行详细介绍。下面举例说明如何使用输入／输出流技术来实现与例3.1完全相同的功能。

**例3.2** 演示基本的C++输入／输出流的使用方法。

```
#include <iostream.h>
void  main( )
{
  char a;
  cout << "Please input a character:";
  cin >> a;
  cout << "The character is: " << a << endl;
}
```

从上面的例子可以看出，使用输入／输出流时不再需要对输入和输出的变量类型加以说明，系统会自动识别，这一点是利用面向对象的重载技术实现的，是对传统的输入／输出函数的重大改进。下面我们来具体介绍输入流的基本使用方法。

在程序执行期间，使用cin来给变量输入数据，其一般格式为：

```
cin >> <变量名1>  [ >> <变量名2> >> ... >> <变量名n> ]
```

其中，运算符“>>”是用来从操作系统的输入缓冲区中提取字符，然后送到内存变量中。在C++中这种输入操作称为“提取”（extracting）或“得到”（getting），因此运算符“>>”通常称为提取运算符，表示将暂停程序的执行，等待用户从键盘上输入相应的数据。

输入流的使用方法记忆起来非常简单，可以将cin想象成为键盘，提取运算符“>>”想象成为指向箭头。由于“箭头”是从cin“指向”内存变量的，因此代表了从键盘获得数据，传入到内存变量中。

在C++中，运算符“>>”具有多种功能，它除了可以做输入流的提取运算符之外，还可以作为位运算的右移运算符来使用。但是，当它和cin关键字连用时，它只能作为提取运算符来使用。这种同一个符号具有不同的使用含义的特性是C++对传统的C语言的一种重要扩充，利用了面向对象编程技术中的运算符重载技术，具体的细节将在本书后面的有关运算符重载部分加以详细说明。

根据上面的格式说明，在每个提取运算符后面只能跟一个变量名，但“>> <变量名>”的组合可以重复多次。即在cin后面使用一次“>> <变量名>”可以给一个变量输入数据，在cin后边重复使用多次“>> <变量名>”可以一次给多个变量输入数据。例如，假设有变量说明：

```
int        a, b;
float      c, d;
char       e, f;
```

在程序执行期间，要求把从键盘上输入的数据送给以上六个变量时，可以用cin按照如下方式来实现：

```
cin >> a >> b;      //A
cin >> c >> d;      //B
cin >> e >> f;      //C
```

当执行到A行语句时，因为有数据流提取运算符“>>”存在，程序停止运行，等待用户从键盘上输入数据。假设这时用户输入：

```
100    200 <Enter>
```

则输入流操作符会根据顺序和变量的类型将100赋给整型变量a，将200赋给整型变量b。其中，<Enter>表示回车键。在使用cin接收数值数据时（包括整型数据和实型数据），为了区分排列在一起的若干个数据，需要在相邻的数据之间加入分隔符。这里的“分隔符”在C++中具有特定的含义，主要包含三种：<Space>（Space键）、<Tab>（Tab键）和< Enter>（回车键）。一般在输入的数值数据之间可以用一个或多个分隔符隔开，系统在进行处理时，会将相邻的多个分隔符视为一个来进行处理。

由于三种分隔符在作用上是等价的，因此用户可以任选一种分隔符来输入数值数据。输入的方式也就跟着变化了，上面用户的输入方式也可以改为如下方式：

```
100 <Enter>
200 <Enter>
```

或者

```
100  <Tab>  200 <Enter>
```

当前面的程序执行到B行时，又开始了一个新的输入流操作，又重新开始等待用户从键盘上输入数据。只不过这时提取操作符后面的变量类型变成了实型数据类型。假设用户输入：

```
3.141593       77 <Enter>
```

则将3.141593赋给了实型变量c，将77.0赋给了实型变量d。提取数据的方法和原理与A行完全相同。

在数值数据被分隔符区分以后，如果要使得计算机接收数据，在每行数据输入完毕后还要输入回车键加以确认。在每行末尾输入回车键的作用是：

（1）告诉cin一行数据已经输入完毕，cin开始提取用户输入的数据（忽略分隔符），并依次将所提取的数据赋值给cin中所列举的变量中尚未获得数据的变量。

（2）在屏幕上显示光标换行，为下一行的输入或者显示程序的输出结果做好准备，起到输入数据之间以及输入数据与输出结果之间的分隔符的作用。

当cin遇到回车键时，如果用户输入的数据与输入语句中等待获得数值的变量的个数不等，会出现两种情况的处理：

（1）用户输入的数据的个数小于变量的个数。则此时在提取完输入行的有效数据后仍有变量没有获得数值，当前的输入语句不会结束，会继续等待用户输入新的一行数据。当用户继续输入数据并且按回车键确认输入后，cin输入语句会继续提取新的输入行中的数值给尚未获得数值的变量，直到当前输入语句中的所有变量都获得了具体的数值为止。在上面的例子中，直到用户输入了两个数值100和200，A行的a变量和b变量分别获得数值后，A行才执行结束。否则程序将会一直等待用户继续输入数值。

（2）用户输入的数据个数大于变量的个数。则输入语句只依次提取输入行中的部分数值给变量。而多余的数值会被下一条cin输入语句中的变量所提取，或者在没有后续的输入语句的情况下被程序舍弃。

假设在执行A行时，用户输入：

```
100      200      3.141593      77 <Enter>
```

A行的cin会将输入行中的100和200分别赋给变量a和b，这时A行中的两个整型变量都获得了有效的数据，A行执行完毕；接着执行B行，因为用户输入行中的数据没有被A行提取完，于是B行接着提取，把用户输入行的后两个数据3.141593和77.0分别赋给实型变量c和d。根据上面的例子可以总结出：在输入数值数据时，只要按照分隔符和行末回车符的规则来输入，可以将四个输入数据在一行内输入，在两行内输入，或者分多行输入。

如果用户输入的一行仅仅是一个回车键，则cin把该键作为分隔符来处理（将其忽略），而不是行末回车符。这时没有数据输送给变量，只是在屏幕上显示光标换行。程序会继续等待用户输入数据，直到用户输入有效数据为止。

在cin输入流的使用过程中，程序员不需要指定输入的数据类型，输入数据的类型由cin根据变量的类型来自动判定。因此，从键盘上输入数据的个数、类型及顺序，必须与cin中列举的变量一一对应。若输入的类型不对，则输入的数据不正确。例如：

```
int  a, b;
cin >> a >> b;
```

执行cin时，若输入字符数据：

```
D     F <Enter>
```

则由于用户输入的数据是字符类型，而对应的接收变量是整数类型，两者类型不匹配，变量a得不到有效值，其值为0。而由于前面已经出现错误，则后续的变量b无法继续获得数值，其值为一个随机数。同时，错误会继续向后延伸，造成后续的cin语句也不能正确提取数据。

再如：

```
int  a;
float  b;
cin >> b >> a;
```

执行cin时，若输入：

```
300     1.234 <Enter>
```

则根据第2章学习的内容，实型b变量进行数据类型转换，将整数300转换为实数300.0，最终获得300.0的数值，影响不大。而整型变量a也同样进行数据类型转换，将小数1.234转换为整数1，小数部分被丢弃。这样可能会造成后续的、有变量a参与计算的计算结果产生极大的偏差。

上面我们以数值数据的输入为例，介绍了cin输入流的基本使用方法。在输入整型数据时，除了标准的输入方法以外，还有一些其他格式的输入方法，并且字符数据输入的写法也与数值数据输入的写法有些不同。下面我们就这些不同点，分小节分别加以具体说明。

### 3.2.1 输入十六进制或者八进制数据

对于整型变量，从键盘上输入的数据默认为十进制。除此以外，还可以按照八进制或者十六进制来输入数据。如前面例子中所示，在缺省的情况下，系统规定输入的整型数都是十进制数据。当要求按照

八进制或者十六进制输入数据时，必须在cin中指明相应的数据类型：hex表示十六进制，oct表示八进制，dec表示十进制。具体应用，如例3.3所示。

**例3.3** 演示不同进制的输入／输出流使用方法。

```
#include <iostream.h>
void  main( )
{
    int  a,b,c,d;       // 变量名a,b,c,d
    cin >> hex >> a;    // 指明输入为十六进制数
    cin >> oct >> b;    // 指明输入为八进制数
    cin >> c;           // 输入仍旧是八进制数
    cin >> dec >> d;    // 指明输入为十进制数
    cout << a <<','<< b <<','<< c <<','<< d << endl;
    }
```

当执行到语句cin时，若输入的数据为：

```
20  21  22  23 <Enter>
```

则根据程序的进制设定，将第一个数20作为十六进制数赋值给变量a、第二个数21和第三个数22作为八进制数分别赋值给变量b和c、将第四个数23作为十进制数赋值给变量d。无论将整数按照什么进制输入变量，在变量中都是按照二进制数的方式存放并且按照十进制数方式显示的。在输出的时候如果不加特别说明，系统会按照十进制的方式进行输出。如在例3.3中针对上述输入的输出结果为：

```
32, 17, 18, 23
```

其中，十进制的32对应的是十六进制的20；十进制的17和18对应的是八进制的21和22；最后一个变量原来就是按照十进制形式输入的，因此依然是23，没有变化。

在C++中，可以用加0x前缀的方式将数值常量表现为十六进制的形式，用加0前缀的方式将数值常量表现为八进制的形式。因此，按照十六进制或八进制的形式输入变量的数值，可以直接与十六进制或八进制形式的常量混合运算。在某些特殊的运算中（例如位运算等），直接使用十六进制或八进制数据进行运算可以得到更加清晰的结果。要做到直接使用十六进制或八进制数据进行输入，必须在cin中指明输入数据时所用的数制。

使用非十进制输入数据时，要注意以下几点：

（1）八进制或十六进制数的输入，只能适用于整型变量，不适用于字符型变量、实型变量。

（2）当在cin中指明使用的输入数制后，则所指明的数制一直有效，直到在下一个cin语句中指明采用不同的输入数制时为止。例如在上例中，输入c的值时，虽然没有再次指定按照八进制输入，但程序仍然会按照八进制进行输入。

（3）用户从键盘输入数据的格式、个数和类型必须与cin中所列举的变量类型一一对应。一旦输入出错，不仅使当前的输入数据不正确，而且使得后面的提取数据也不正确。具体的例子，请参考上面对输入流基本操作的介绍。

使用输入流输入数据时，如果出现输入错误，可以通过相关函数来检测错误，并且可以根据具体的错误进行相应的处理。

### 3.2.2 输入字符数据

当要为字符变量输入数据时，输入的数据必须是字符型数据。此时cin的格式及用法与输入十进制整数和实数时是相同的。设有如下的程序片段：

```
char c1, c2, c3, c4;
cin >> c1 >> c2 >> c3;
```

当执行cin语句时，cin等待用户从键盘上输入数据。如果输入：

```
A  b  C  <Enter>
```

则cin分别将字符A、b、C赋值给字符型变量c1、c2、c3。如果输入:

```
AbC  <Enter>
```

cin也同样分别将字符A、b、C赋值给字符型变量c1、c2、c3，即字符之间是否有分隔符都一样。在缺省的情况下，cin会自动过滤输入的空格等分隔符（参考输入流基本功能介绍中的分隔符概念），输入流cin不会将输入的空格和回车键等分隔符赋给字符型变量。

在使用输入流输入字符数据时，最容易出现的错误是将数字与字符数据混淆，如例3.4所示。

**例3.4** 演示输入／输出流的字符输入方法。

```
#include <iostream.h>
void  main( )
{
  int  a, b;
  char c, d;
  cin >> a >> b;
  cin >> c >> d;
  cout << a <<','<< b <<','<< c <<','<< d << endl;
}
```

假设用户输入如下数据:

```
12  34  5678 <Enter>
```

则由于变量a和变量b是整数类型，输入流cin会根据分隔符的标识，将12赋值给变量a，将34赋值给变量b。接下来就是给字符类型变量c和变量d赋值了。根据第2章学习的知识，我们知道一个字符占用一个字节的空间，一个字符变量存放一个字符。因此，虽然5678是连写在一起的数字，但是输入流不会将它们当成一个整体赋值给字符变量，而是将它们看成四个连续的字符。'5'字符将被赋值给变量c，'6'字符将被赋值给变量d，而剩余的'7'字符和'8'字符由于没有对应的变量，将被舍弃。

在进行字符输入时，不但可以按照顺序输入字符，还可以对输入的内容有选择地接收，这时就要用到函数cin.ignore( )了。cin.ignore( )的作用是在字符读取过程中忽略若干个字符，而读取后面的字符。其格式为:

```
cin.ignore( <忽略的字符个数> );
```

例如将程序例3.4中的输入一段修改如下:

```
cin >> a >> b;
cin.ignore(3);
cin >> c >> d;
```

同样输入如下数据:

```
12  34  5678 <Enter>
```

但这时的输出结果却变化如下:

```
12, 34, 7, 8
```

可以看出，cin.ignore(3)语句让程序忽略了整数34后面的三个字符（包括空格符），而直接从第四个字符读起。将'7'字符赋值给了变量c，将'8'字符赋值给了变量d。

前面提到，输入流cin不会将输入的空格和回车键等分隔符赋给字符型变量。由于分隔符也是有效字符，那么当用户想要将三种分隔符作为字符输入计算机时，如何实现呢?

借助于函数cin.get( )可以实现上述的要求。函数cin.get( )的作用是把从键盘上输入的每一个字符，包括空格符和回车符等分隔符都作为一个输入字符赋给字符型变量。其格式为:

```
cin.get( <字符型变量> );
```

函数cin.get( )从输入行中取出一个字符，并将它赋值给字符型变量。该语句一次只能从输入行中提取一个字符。例如:

```
char c5, c6, c7, c8;
cin.get(c5);
cin.get(c6);
cin.get(c7);
cin.get(c8);
```

当程序执行第一行输入语句时，如果用户输入:

```
A B <Enter>
```

在输入字符A前没有空格，在字符A与B之间有一个空格，则根据前面的介绍，程序会将字符A、空格、字符B分别赋值给变量c5、c6、c7；第四行语句中的c8变量也将获得字符值：回车字符。也就是对于cin.get( )来说，用户一共输入了4个字符。若第四行使用语句:

```
cin >> c8;
```

代替cin.get(c8)，则cin将回车作为确认符和分隔符处理，在屏幕上显示光标换行，但并不会将回车符存入变量c8。这时由于变量c8尚未获得有效值，程序会一直等待用户继续输入一个有效字符。

## 3.3 cout

与cin输入流对应的是cout输出流，使用cout输出流可以方便地将内存中的数据显示给用户察看。根据本章开始的介绍，在程序中输出功能的重要性还要大于输入功能，任何一个有实际用途的程序至少要有一个输出来显示计算结果。因此，在本节中将花较多的篇幅来介绍把字符、整数、实数及字符串输出到显示器的基本方法。同样，有关输出流cout所能实现的详细功能，将在第14章中进行介绍。

在程序执行期间，我们使用cout将变量中的数据或者字符串输出到屏幕上或者文件中，以便用户察看。其一般格式为:

```
cout << <表达式1>  [ << <表达式2> << ... << <表达式n> ]
```

其中，运算符“<<”是用来将内存中的数据插入到输出流cout中，然后输出到标准输出设备——显示器上的。在C++中这种输出操作称为“插入”(inserting)，因此运算符“<<”也称为插入运算符，它将紧跟其后的表达式的值插入到输出流cout中，输出到显示器当前光标的位置。程序执行中如果遇到插入运算符，则将内存变量的值输出，然后程序继续向下运行。

输出流的使用方法记忆起来也非常简单，可以将cout想象成为显示器，插入运算符“<<”想象成为指向箭头。由于“箭头”是从内存变量“指向”cout的，因此代表了将内存变量中的数据插入到输出设备中去。

与提取运算符“>>”类似，运算符“<<”具有多种功能，它除了可以做输出流的插入运算符之外，还可以作为位运算的左移运算符来使用。但是，当它和cout关键字连用的时候，它只能作为插入运算符来使用。这同样是利用了面向对象编程技术中的运算符重载技术，具体的细节将在后面的有关运算符重载部分加以详细说明。

根据上面的格式说明，在每个插入运算符后面可以跟一个表达式。在显示数据时可以将变量值和字符串融合在一起，构造出直观的答案。这样，输出流cout在使用中的变化就要比输入流cin来得复杂，同样能够实现的功能也要强大一些。同样，“<< <表达式>”的组合可以重复多次，即在cout后面使用一次“<< <表达式>”可以输出一个表达式的值，在cout后边重复使用多次“<<<表达式>”可以一次输出多个表达式的值。并且可以将多个变量的值与多个字符串组合在一起，构成复杂的输出结果，如例3.5所示。

**例3.5**　演示使用输出流技术的问答式程序。

```
#include <iostream.h>
```

```
void  main(void)
{
     cout << "请输入变量i的值:";
     int  i;
     cin >> i;
     cout << "您输入的数值是: " << i <<endl ;
}
```

在执行第一条cout语句时，在显示器上显示:

```
请输入变量i的值:
```

即cout将双引号中的字符串常量按其原样输出。接着执行cin语句，屏幕上的光标停留在冒号的后面，等待用户输入变量i的值。通常在程序设计中，在每一个cin语句之前，都会用一个cout语句给出提示信息，指明用户给什么变量输入数据，并且以什么样的数制输入。这样用户面对的不再是孤零零的黑色屏幕，而是针对每一个提示输入相关的数据，能够降低出错的机会。这种程序的编制方法称为问答式程序界面。

下面再举例说明输出流cout对多变量的处理方法，如例3.6所示。

**例3.6** 演示使用输出流技术的多变量输出。

```
#include <iostream.h>
void  main( )
{
     int  a=10, b=20, c=30, d=40;
     double  m=5.23, n=100;
     cout << a << b << endl;
     cout << c << n-d << endl;
     cout << m << n << endl;
}
```

输出结果为:

```
1020
3060
5.23100
```

以上每一个cout语句输出一行，其中endl表示要输出一个换行符，它等同于转义字符'\n'。endl是短语“end of line”的缩写。当用cout输出多个数据时，缺省情况下，是按每一个数据的实际长度输出的，即在每一个输出的数据之间没有分隔符。显然，如果直接输出数据，所有的数据将连接在一起，无法分清哪一个变量的输出值是多少。如第一行输出1020，实际上是先输出a的值10，再输出b的值20。为了区分输出的数据项，在每一个输出数据之间要输出分隔符。分隔符可以是空格、标点符号或者换行符等。如上面的输出语句可改写为:

```
cout << a << ',' << b<< endl;
cout << c << ',' << n-d << endl;
cout << m << ',' << n << endl;
```

则输出结果为:

```
10, 20
30, 60
5.23, 100
```

还可以改写成:

```
cout <<"a=" << a << '\t' <<"b=" << b << endl;
cout <<"c=" << c << '\t' << n << "-" << d <<"=" << n-d << endl;
cout <<"m=" << m << '\t' <<"n="<< n << endl;
```

则执行这三个输出语句后，输出:

```
a=10      b=20
c=30      100-40=60
m=5.23    n=100
```

从上面的例子可以看出，在cout中输出数据的格式并不是根据变量的类型确定的。例如：变量y是双精度浮点数类型，按道理应该输出一个小数格式的数据，但是输出的数据却是一个整数100。在输出流cout中是根据变量的数值而不是根据变量的数据类型来决定输出数据的格式的。同时，我们还要看到，一个清晰的输出结果往往是由字符串、控制字符和变量值共同组成的。所以在使用输出流cout时，一定要综合使用多种控制技术，以便用户能够从输出结果中轻松地获得需要的信息。

使输出的数据项之间隔开的另一种办法是使用setw( )函数来指定输出数据项的宽度。例如上面的三个输出语句可以改写为:

```
cout << setw(10) << a << setw(10) << b << endl;
cout << setw(10) << c << setw(10) << n-d << endl;
cout << setw(10) << m << setw(10) << n << endl;
```

其中，setw(10)指明其后的输出项占用的字符宽度为10，即括号中的值指出紧跟其后的输出项占用的字符位置个数，并且向右对齐。setw是“set width”的缩写。执行以上三个语句后的输出为:

```
10         20
30         60
5.23        100
```

使用setw( )函数应该注意以下四点:

(1) setw( )函数是定义在iomanip.h头文件中的系统函数，所以要使用它就必须在程序的开始位置包含头文件iomanip.h，即在程序的开头增加:

```
#include  <iomanip.h>
```

(2) 括号中必须给出一个正整数或者数学表达式（值为正整数），它指明紧跟其后输出的数据项的宽度。

(3) 该设置仅对其后的一个输出项有效。一旦按指定的宽度输出其后的输出项后，程序又自动回到原来的按实际宽度输出的缺省输出方式。

(4) 当设置了数据的输出宽度后，如果数据的实际位数小于指定的宽度，则添加填充符。如果数据的实际位数大于指定的宽度，则数据按照实际的宽度输出，不会按照指定的宽度来截断数据。

上面介绍了输出流cout对于数据的通用处理方法。在cout中除了基本操作方法以外，还针对不同的数据设计了一些特殊的处理方法。下面分别按照数值数据和字符数据来对这些处理方法进行分类介绍。

### 3.3.1 输出八进制数、十六进制数和用科学计数法表示的数

与上一节提到的cin中的处理类似，在cout中对于整型数据可以指定以十六进制或八进制输出，对于实型数据可以指定以小数形式或者科学计数法形式输出，如例3.7所示。

**例3.7** 演示使用输出流技术按照特定的格式输出数值数据。

```
#include <iostream.h>
void main(void)
{
     int  a=10, b=20, c=30, d=40;
     double  m=5.23, n=100;
     cout << "a=" << oct << a << '\t' << "b=" << b << endl;
     cout.setf(ios::scientific, ios::floatfield);          // 按照科学计数法输出实数
     cout << "c=" << hex << c << '\t' << "n-d=" << n-d << endl;
     cout << "m=" << m << endl;
     cout.unsetf(ios::scientific);                         // 取消按照科学计数法输出
```

```
    cout << "m=" << m << '\t' << "n=" << n << endl;
}
```

执行该程序后，输出:

```
a=12    b=24
c=1e    n-d=6.000000e+001
m=5.230000e+000
m=5.23  n=100
```

在程序中，hex和oct这两个词用在输出流cout中，分别代表了程序应该按照十六进制和八进制格式输出数据。与cin中的用法类似，一旦在cout语句中指定了输出的进制格式后，这种格式将一直有效，直到指定另外一种格式为止。

在程序中调用了两个系统函数: cout.setf( )和cout.unsetf( )。其中，cout.setf( )函数用来指定对实数按照科学计数法格式输出，而cout.unsetf( )用来终止对实数按照科学计数法格式输出。与整数的特定进制格式输出相同，一旦指定了按照科学计数法输出实数，其后所有的实数都将按照科学计数法的格式输出，直到取消科学计数法的输出格式为止。关于cout.setf( )函数和cout.unsetf( )函数的使用原理，我们将在第14章中进行介绍。

在例3.7输出的结果中，由于指定了按照八进制格式输出变量a和变量b的值，因此将10输出成12，将20输出成24。随后指定将实数按照科学计数法格式输出，由于变量c是整数，因此将30按照十六进制格式输出成1e。由于变量n是实数，因此计算结果也是实数，按照科学计数法格式输出成6.000000e+001。随后，按照科学计数法格式输出变量m的值为5.230000e+000。取消了科学计数法格式以后，输出变量m和变量n的值分别为5.23和100。

### 3.3.2 输出字符或字符串

输出流cout除了能够输出正常可视的标准字符外，还可以输出一些不可见的控制字符，这些字符就是第2章中所说的转义字符。如下面的程序片断所示:

```
char  c='a', c1='b';
cout << "c=" << c << '\t' << "c1=" << c1 << '\n';
```

执行cout语句时，先输出“c=”可视的标准字符串；接着输出变量c的值；再输出不可视的“横向制表符”控制字符，跳到下一个Tab位置；再输出“c1=”可视的标准字符串；接着输出变量c1的值；最后输出一个不可视的“换行符”控制字符，表示以后的输出从下一行开始。所以该行的输出结果为:

```
c=a     c1=b
```

在计算机中将屏幕位置按照8列为一个单位进行划分，每8列就是我们所说的一个Tab位置。当用户按下键盘上的Tab键时，光标会从当前的8列区间移动到下一个8列区间的第1列的位置。例如: 当前光标处于屏幕的第3列属于第一个区间，按下Tab键，光标会移动到第9列，即第二个区间的第1列。如果光标处于第7列，因为同样属于第一个区间，按下Tab键，光标同样会移动到第9列。同样的道理，如果光标处于第12列，则按下Tab键，会移动到第17列。以此类推，可以很容易地计算出输出内容之间的间隔。例如: 在上面的例子中两个输出变量之间的间隔为5个字符。

根据上例所示，使用第2章介绍的转义字符的书写方法，用cout可以输出任何ASCII码的字符。

## 3.4 总结

本章简单介绍了Visual C++中输入／输出流的使用方法和技巧。通过本章的学习，读者可以使用输入／输出流技术，根据不同的情况灵活使用cin和cout的各种格式和函数，实现程序完善的输入和输出效果。

输入流cin是根据变量的数据类型来决定接收的数值的。因此，在使用输入流技术输入数据时，输入值的顺序和类型必须要和输入流语句中的变量的顺序和类型一致。否则，就得不到正确的数值。

在输入流中，回车符既是分隔符又是确认符。因此，在使用回车符时，要特别注意分辨回车符的作用，掌握回车符的正确使用方法。

输出流cout不是根据变量的数据类型来决定输出的数据格式，而是根据变量中具体的数值来决定输出格式。因此，实型变量也可以输出整数格式的结果。

使用setw( )可以指定输出数据项的宽度。不过需要注意的是，当数据项的实际宽度大于指定宽度的时候，系统会按照实际宽度输出。

## 习题3

1. 设有如下语句:

```
int  x, y;
char a, b;
cin >> x >> a >> y >> b;
```

如果希望执行完cin语句后，x=3，y=5，a='7'，b='8'。则用户应该如何书写输入的数据?

2. 若用户输入的数据如下:

```
12  a  30  345
```

程序中有变量a，b，c，d，接收用户输入的数据后，四个变量的值分别为a=2，b=10，c=30，d=5。请编写程序实现本题所描述的功能。

3. 请编写一个程序，每次从键盘接收5个字符，统计这5个字符中空格的数目并且在屏幕上输出统计的结果。

4. 设有如下语句

```
float x=100, y=3.14;
char a='\t ', b='s';
cout<<"x is" << setw(10) << x <<'\t';
cout.setf(ios::scientific,ios::floatfield);
cout<<"y="<< y << endl;
cout<< a << b <<endl;
```

则最终的输出结果是什么?

5. 设有变量定义如下:

```
int  i=8, j=12;
double  x=3.14, y=90;
```

希望得到如下的输出结果:

```
i=10    j=c
x=3.140000E+000    y=90
```

请编程实现。

6. 请编程提示并接收用户输入的三角形的三条边长，然后用三边面积公式计算三角形的面积，最后输出求得的面积值。

计算方法：设$a$、$b$、$c$为三个边长，$s=\frac{a+b+c}{2}$，

面积：Area=$\sqrt{s\times(s-a)\times(s-b)\times(s-c)}$。

# 第4章 C++的流程控制

计算机程序可以实现各种复杂的运算，这不是仅仅靠前面介绍的输入／输出以及基本的数据操作就可以完成的。在程序中真正实现各种实际运算功能的是程序的算法，学习计算机编程最重要的也是学习算法的设计方法。只要掌握了算法的设计原则，能够使用算法设计工具来熟练地设计出算法，那么无论是用何种编程语言都可以设计出优秀的程序。

本章介绍程序的算法原理、算法设计规则、算法的表示工具和程序的设计方法，并且结合C++的基本程序设计语法，介绍如何使用Visual C++来设计算法和实现实际的应用程序。

## 4.1 算法概述

所谓程序的算法，就是使用程序解决问题的计算步骤。我们在日常生活中无论做任何事情都是有步骤的，例如出外旅游首先要到旅行社报名，签订旅游合同，付款，按时到指定地点出发，到各个风景点游玩，最后回家。这其间所完成的步骤不能颠倒和出现差错，否则事情就无法顺利完成。

### 4.1.1 算法的作用和类别

当我们使用计算机去解决问题时，与做一件日常生活中的事情没有两样，也需要按照特定的步骤来完成。由于计算机主要是以数学计算的方式来解决问题的，因此我们以解数学题的例子来介绍程序算法。想象一下我们人类是如何去解决一道数学题的：首先阅读题目，理解题意；然后利用我们大脑中已有的知识去组合列出解题的公式；最后将数据代入公式求得数值解。其中在公式中规定了解题的步骤，先做什么计算后做什么计算，这些次序是不可以颠倒的，否则就得不到正确的结果。

程序的设计过程与我们解决数学问题的过程一样，其中程序的算法设计对应的就是上述列出解题公式的操作。解题公式规定了数学计算的步骤，保证取得正确的计算结果，程序算法规定了计算机程序的运行步骤，也保证了程序的运行可以取得正确的结果。解题公式可以简单或者复杂，程序算法也可以简单或者复杂。

从数学题的解法中可以看到，列出公式与数值的计算是分离的。同样的道理，在计算机程序中算法与数据也是分离的。一般认为，一个计算机程序包括两个部分：程序和数据。所谓程序就是对操作步骤的描述，也就是描述算法（algorithm）。所谓数据就是在程序的运行过程中被用来计算的数值，这些数值在计算机中是按照一定的数据结构（data structure）来存放的。

前面提到，计算机算法可以等同于日常生活中人们处理事情的行动步骤，这就可以推演出一个算法效率的问题。不同的人在解决同一个问题时会根据自己的生活经验合理地安排行动步骤。有的人安排的行动步骤非常合理，我们就说他的办事效率比较高；有的人安排的行动步骤不太合理，我们就说他办事没有效率。在计算机程序设计中，这种问题也同样存在。对于同一个问题，可以有不同的解题方法和步骤。采用优秀的解题方法，合理安排计算步骤，以最少的计算步骤完成计算任务的方法我们称为高效率算法。同理，使用了比较笨拙的解题方法，通过较多的运算步骤来实现同样的计算任务的计算方法我们称为低效率算法。

例如，求1 + 2 + 3 + … + 99 + 100，即 $\sum_{n=1}^{100} n$ 。可以直接计算，即先计算1+2，然后将计算结果再加上3，以此类推，一直加到100。也可以使用一些技巧性的算法，例如：$\sum_{n=1}^{100} n$ = (100+1)+(99+2)+(98+3)+…+(51+50)= 101 × 50 = 5050。此外还可以有其他多种计算方法。

从上面的例子可以看出，同样正确地解决一道计算题，在第二种算法中只进行了50次加法和1次乘法，总共51次计算，而在第一种方法中，总共进行了99次加法，显然第二种计算方法的效率要优于第一种计算方法。因此，我们说第二种计算方法是高效率算法，而第一种计算方法是低效率算法。

一般来说，在程序设计中希望采用简单的和运算步骤少的方法。因此，为了有效地进行解题，不仅需要保证算法正确，还要考虑算法的质量，选择合适的算法。同时，衡量一个程序员是否具有良好的软件开发能力，也往往考察其是否能够在解决实际问题时设计和使用合适的算法。

### 4.1.2 算法的设计原则

算法控制程序运行的步骤，以便获得需要的运算结果。但不是任何一种随意书写的算法都能够获得运算结果，算法的设计必须遵循以下几个原则。

#### 1. 符合数学计算规则

由于计算机是按照数学规则来进行计算的机器，只有符合数学规则的计算步骤才可以被计算机正确执行。如果在算法中书写了违反数学规则的命令，那么重则程序出现错误，终止运行，轻则会出现一些莫名其妙的计算结果。例如，两个变量a=3，b=0，如果书写公式a/b，则计算结果是无穷大，而计算机无法表示无穷大，这时程序出错，终止运行。再例如，变量c=－5，如果调用函数sqrt(c)求变量c的平方根，由于负数的平方根非实数，计算获得无效的答案。

#### 2. 保证结果确定

一个正确的算法中的每一个步骤获得的计算结果都应当是确定的，只有这样才能最终获得确定有效的计算结果。如果一个算法对同一组数据进行多次计算，竟然获得多个不同的结果，这种算法是不确定的，是无效的。保证算法的确定性，最重要的方面是排除程序中随机数的产生。在C语言和C++中，很多变量在没有初始化的时候，其中存放的值就是随机数。如果将这些存放着随机数的变量拿来直接使用或作为逻辑条件来判断，将使计算结果不确定或造成整个程序的逻辑混乱。因此，程序编写的过程中一定要注意变量的初始化问题，使得算法的执行过程唯一。

#### 3. 程序能够正常结束

一个合理的算法应包含有限的操作步骤，而不能是无限的。在后面的章节中我们会介绍一种程序设计的重要结构——循环结构，使用循环结构可以使得程序按照一定的数学规律，利用迭代算法对一组数据进行计算。但是在设计使用循环结构的算法时，如果不能正确地处理逻辑判断条件，则循环将永远不会停止。这种程序永远也不会得到结果，因此是无效的。

还有一种情况，由于对循环结构的控制不合理，虽然程序最终可以运行结束并且获得结果，但是可能需要10年、20年，甚至于100年、1000年。我们认为这样的算法效率太低，没有实际的使用价值，也是无效的。

一个程序能够在有限的时间内计算获得正确的结果，才算是一个能够正常结束的有效程序。对于复杂的科学计算，一般要求能够在几个月的时间内算完。对于复杂的商业计算，一般要求能够在24小时以内算完，而一般的商业计算则要求瞬间完成。

#### 4. 合理的输入

输入／输出也是算法设计中需要考虑的一个重要因素。根据第3章的介绍，一个实际有效的程序中应该含有零个或者多个输入。如果程序预先将所有需要处理的数据都嵌入到程序中，则在程序的运行过程中不再需要用户输入数据，这时程序的算法中可以不加入输入操作。在大多数情况下，为了保证程序的灵活性，一般都会在程序的算法中加入一个或者多个输入。

#### 5. 合理的输出

任何实际程序的运算目的都是希望求得运算结果并且显示给用户，如果计算的结果只有计算机自己知道是没有意义的。所以，一个有效的程序在设计算法时必须保证程序至少要有一个输出。根据不同程

序的要求，算法中也可以实现多个输出。

### 4.1.3 算法的表示工具

算法是程序的计算步骤，就像在解数学题目时要用数学公式来表示计算的步骤一样，在设计程序算法的时候也需要一种表示方式，用来描述算法的实施步骤。

在实际应用中我们有四种常用的表示方法来表示一个算法：自然语言、流程图、N-S图和伪代码。其中伪代码使用的频率不高，这里我们不作介绍。

自然语言就是人们日常生活中使用的语言。用自然语言表示算法通俗易懂，但是编写和修改的工作量较大，而且由于语言自身的特点，容易造成误解，往往需要根据上下文来判断句子的真正含义。同时，由于程序员在使用自然语言描述算法时，往往喜欢使用母语，这也不利于国际间的开发合作。

流程图是目前全球软件开发领域使用最广泛的算法表示工具，它通过一些严格定义图形的组合来表示算法的步骤以及数据变化的走向。用图形表示算法，直观形象，易于理解。由于流程图中大量使用图形和英文符号，几乎不会出现程序员的母语，因此通过流程图，使用不同母语的程序员也可以方便地交流算法设计的思想。目前全球广泛使用的流程图的标准是由美国国家标准化协会ANSI（American National Standard Institute）规定的标准。其中一些常用的流程图符号如图4-1所示。

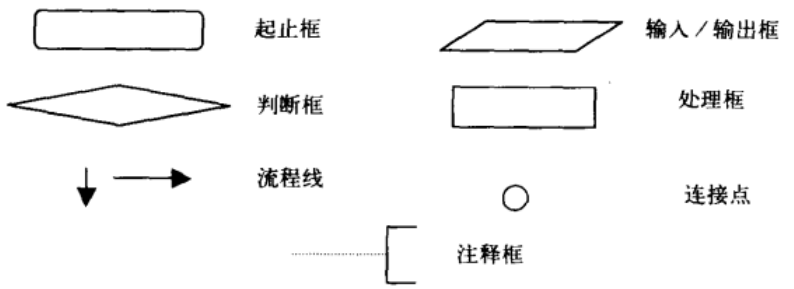



图4-1 流程图中的常用符号

图4-1中起止框是一个圆角矩形，它的作用很简单，说明程序的起始点和终止点。

输入／输出框是一个平行四边形，算法中所有与输入／输出相关联的步骤都要写在这种框中。

菱形代表判断框，它的作用是对一个给定的逻辑条件进行判断，根据判断的结果是真或者假来决定如何执行后面的操作。它有一个入口，两个出口，具体操作见后续章节中的程序示例。

处理框是一个方角矩形，算法中的大部分内容都会写在这种框中，主要是各种数学运算和逻辑运算。

连接点（圆圈）是用于在设计复杂流程图时，将多条流程分支组合在一起的图形。用连接点，可以避免流程线的交叉或者过长，使复杂的流程图变得清晰可读。由于本书只是介绍C++的基本语法和基本算法，不会牵涉到比较复杂的计算流程，因此连接点的应用就不再具体举例介绍。相关内容，读者可以参考其他介绍算法描述的书籍。

注释框只是为了对流程图中某些框的操作做必要的说明，不是流程图中必须书写的部分。如果不是非常复杂的流程图，一般不用。

流程图已经成为目前世界各国程序员普遍采用的算法描述工具，所以它的用法是一个程序员所必须掌握的。

流程图虽然可以很清晰地表达程序员的思想，但是绘制起来比较复杂，其中大部分时间花费在绘制流程线上。1973年美国学者I.Nassi和B.Shneideman提出的一种无流线的流程图，称为N-S图，它省略了流程线，提高了算法描述的效率。

例如：求解多项式 $1\times\frac{1}{2}\times\frac{1}{3}\times\cdots\times\frac{1}{100}$ 的值。

算法描述如下：定义累乘积变量M和分母变量N

算法步骤：（1）M←1，N←1

（2）$M \leftarrow M \times \frac{1}{N}$

（3）N←N+1

（4）如果N≤100，转第(2)步；否则，转第(5)步

（5）输出M的值

（6）结束

上述算法的描述是采用自然语言加伪代码的方式给出的。用流程图对算法进行同样的描述，如图4-2所示。

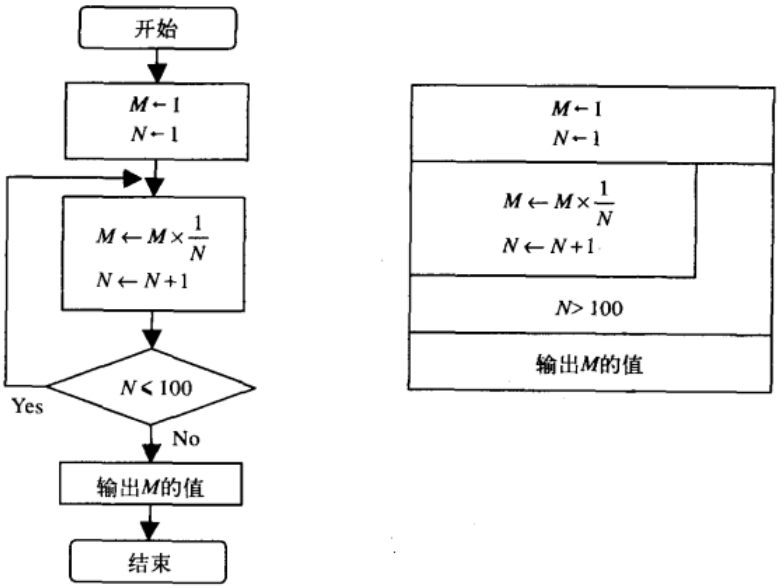



图4-2　流程图（左）和N-S图（右）

### 4.1.4　结构化程序设计中基本结构的表示

在结构化程序设计方法中，人们将所有的计算结构归纳成三种基本结构：顺序结构、选择结构和循环结构。它们的执行流程描述如下：

- 在顺序结构中，先执行A操作再执行B操作，两者次序不能颠倒。其中A和B可以是一条语句，也可以是三种基本结构中的任何一个基本结构，如图4-3所示。

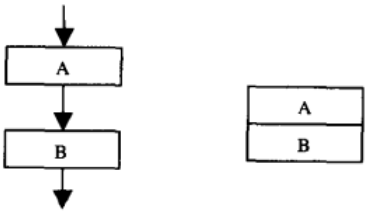



图4-3　顺序结构流程图（左）和N-S图（右）

- 在选择结构中，C代表一个逻辑条件，A和B是两个不同的操作。当C的条件判断为真的时候，执行A操作；当C的条件判断为假的时候，执行B操作。在程序的一次运行中，只能执行A或者B中的一个操作，A操作和B操作是互斥的，如图4-4所示。

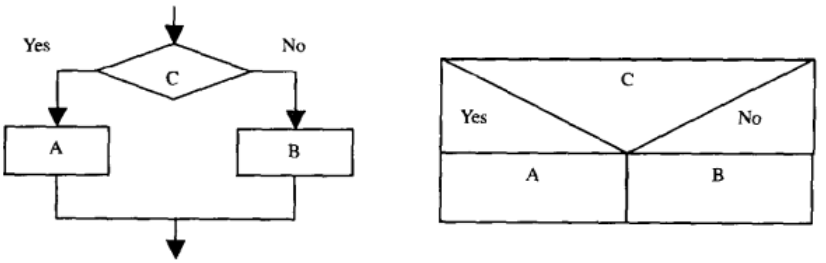
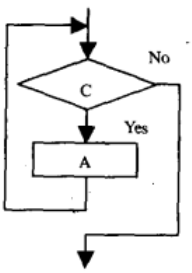



图4-4 选择结构流程图（左）和N-S图（右）

- 基本循环结构又分为两类：当（While）型循环和直到（Until）型循环。这两个名字是根据它们的运算法则来取的。在当型循环中，首先判断逻辑条件C。当C的条件判断为真的时候，执行A操作，并且在执行完A操作后继续进行下一次的条件C判定。当C的条件判断为假的时候，退出循环。由于在循环中执行循环体A操作的条件是“当”C的条件为真的时候，所以称为当型循环。当型循环用流程图和N-S图描述，如图4-5所示。
  - 直到型循环与当型循环的执行顺序有所不同，在直到型循环中，首先执行操作A，然后再判断条件C的值是否为真。如果C的条件判断为假，则再次执行A操作，直到C的条件判断为真为止。当C的条件判断为真的时候，退出循环。由于在循环中是“直到”C的条件判断为真时退出循环，所以称为直到型循环。直到型循环用流程图和N-S图描述，如图4-6所示。



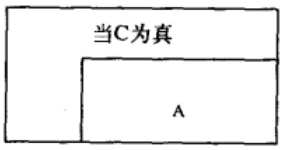
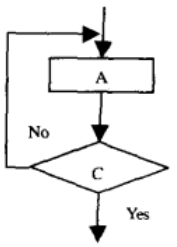




图4-5 当型循环结构流程图（左）和N-S图（右）



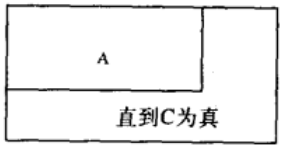




图4-6 直到型循环结构流程图（左）和N-S图（右）

- 当型循环是先判断条件，再按照判断的结果决定是否执行循环体操作。而直到型循环是先执行循环体操作，再进行条件判断，以决定是否继续执行循环体操作。所以在当型循环中，有可能循环体一次都不执行，而在直到型循环中，循环体至少要被执行一次。两类循环在具体执行时存在差别，所以在进行算法设计时，应该选择合适的循环结构。

所有的程序都由这三种基本结构组装而成，所有的算法设计中也都只使用这三种基本结构，这大大简化了程序设计的难度，这种算法叫做结构化的算法。同时，在流程图中也规范了三种基本结构的描述，在以后的流程图中只需要将标准的基本结构进行合理的连接就可以实现算法了。三种基本结构的流程图表示如图4-7所示。

三种基本结构的特点为：

（1）单入口、单出口。在图4-7中虚线框内部是基本结构的运行逻辑，每一个虚线框都只有一个进入和退出的端口。基本结构所具有的这个特点，保证了将多个基本结构组装在一起的时候，程序的运行不会产生歧义性。程序可以沿着一个方向运行，直到结束。

（2）结构中的每一部分都可能被执行到。通过前面对三种基本结构运行逻辑的介绍，我们可以看到，虽然选择结构中每次执行的A或者B部分可能不同，循环结构中的循环体A可能不被执行；但是，这次不被执行的部分，下次还是有可能被执行到的。结构中不会有永远不被执行的无用部分。

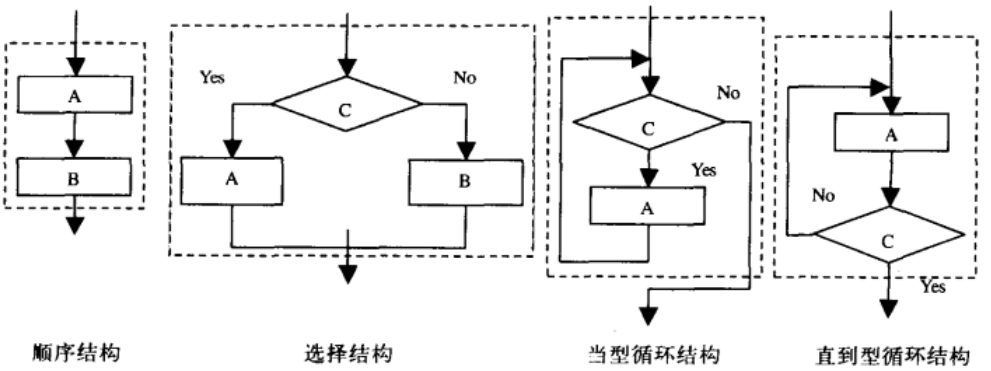



图4-7　程序设计的三种基本结构

（3）没有死循环。在循环结构中，都设定了退出循环的条件，只要满足了这个条件，循环就可以退出，使得程序正常终止。

（4）A、B可能是一个更基本的结构。在三种基本结构中，A部分和B部分不但可以代表基本语句，还可以是基本结构。例如：在一个选择结构中，当条件判断为Yes执行A部分时，A部分可能是一个顺序结构，它又包含了几个顺序执行的语句部分；当条件判断为No执行B部分时，B部分也可能是一个当型循环结构，循环若干次来实现某个计算。

在总结出三种基本结构，开始用基本结构来构造结构化的算法后，我们还需要能够实现三种基本结构的编程工具。于是从20世纪70年代中期开始诞生了一批例如C语言、Pascal语言等能够实现三种基本结构的高级编程语言。我们将这些能够实现三种基本结构的编程语言称为结构化编程语言。

至此，所有实现结构化程序设计的工具都已经准备完毕。但是，真正要用程序去解决一个复杂的问题，光有编程的工具和方法是不够的，还需要有一套先进的程序设计方法。与结构化程序设计工具相配套的是结构化程序设计方法。

## 4.2　C++程序的结构和语句

C++是一种程序编写语言，使用这种语言可以编写出功能各异的程序来。这就好像我们使用一种自然语言就可以写出各种各样的文章来一样。从某种角度来说，我们可以认为程序编写语言与自然语言功能是一样的，都是用来描述事物的工具。在学习自然语言的时候，我们需要学习单词、语法等要素，学习编程语言也一样。不过，由于编程语言中需要记忆的单词非常少，一般只有几十个，语法也很简单，一般只有十几条，不需要特别记忆，因此，编程语言中需要掌握的要素是语句。所有的程序功能都是由一条条的语句来描述实现的，通过对语句的学习，我们可以很轻松地掌握编程语言的关键字和语法。计算机程序运行的时候是将一条条的语句翻译成机器指令，控制计算机产生计算的结果的。所以，语句是计算机程序的基本组成单位。图4-8显示了C++程序的组成结构，以及语句在C++程序中的地位。

一个C++程序可以包含一个或者多个源程序文件。本书介绍基本的C++程序编写方法，所举的例题都非常简单，所以在计算机上验证例题时，只需要一个源程序文件就可以了。但是如果编写复杂的应用系统，特别是编写Windows应用程序，则在一个程序中往往会用到几十个甚至上百个源程序文件。

每个源程序文件中都包含了一些重要的程序段，这些程序段不是直接由语句构成的，而是由若干个函数和预处理命令构成的。函数也是结构化程序设计中的一个重要概念，它将一组语句构造成一个整体，可以在程序中反复使用，这提高了程序的开发、维护效率。关于函数和预处理命令的使用，在后面的第5章和第6章中都有详细的说明。这里需要注意的是，在C++程序中，任何执行语句都必须写在函数中，不允许出现不属于函数的独立执行语句。

函数又由两部分组成：函数说明部分和函数体。其中，函数体中就是我们要编写和存放语句的地方。

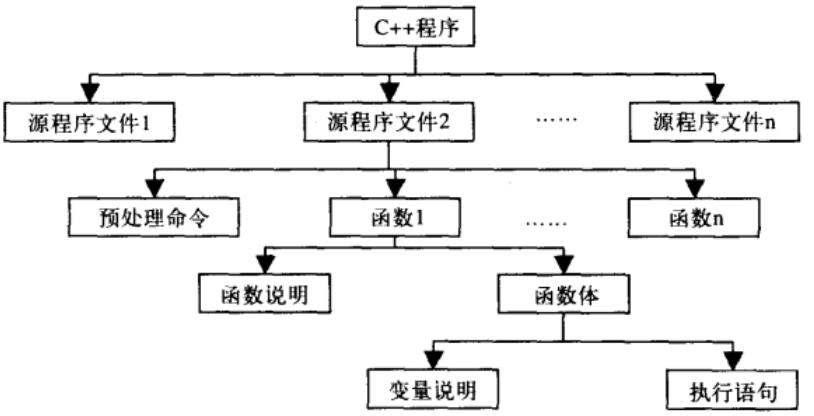



图4-8 C++程序的组成结构

在C++中语句可以分为以下6大类。

### 1. 定义语句

在C语言和C++中，用户可以自己定义数据结构和数据类型，也可以定义变量。这些操作都没有实现数值的运算，尤其在C语言中，它们的定义位置只能放在函数的开始位置，不能与其他语句混用。因为存在这种与其他语句的严格区分，因此在C语言中，定义操作不被称为语句。但是在C++中，这些定义操作可以出现在函数的任何位置，可以与其他任何语句混合使用，因此我们称这类操作为定义语句。

### 2. 控制语句

我们称能够完成一定的控制功能，改变程序执行的方向的语句为控制语句。在C++中，主要的控制语句有如下9种:

| | |
|---|---|
| (1) if( ) ... else ... | 条件语句 |
| (2) for( ) ... | 当型循环语句 |
| (3) while( ) ... | 当型循环语句 |
| (4) do ... while( ) | 直到型循环语句 |
| (5) continue | 结束本次循环语句 |
| (6) break | 中止执行switch或循环语句 |
| (7) switch | 多分支选择语句 |
| (8) go to | 转向语句 |
| (9) return | 从函数中返回语句 |

在C++中，就是通过控制语句来实现前面介绍的结构化程序设计的三种基本结构的。在本章下面的小节中我们将对控制语句的使用方法进行详细的介绍。

### 3. 函数调用语句

在一次函数调用的后面加上一个分号构成的一条语句，实现函数定义中规定的计算（例如: `sqrt(x);`)。关于函数的使用，在第5章将进行详细介绍。

### 4. 表达式语句

在一个表达式的后面加上一个分号构成的一条语句，实现表达式的计算功能。在第2章中已经对表达式的概念作了介绍，两者的差别如下所示:

```
a = b + c              // 表达式
a = b + c;             // 表达式语句
```

### 5. 空语句

仅有一个分号的语句，称为空语句。顾名思义，空语句中什么计算也没有，什么操作也不做，如下

所示:

```
;  // 空语句
```

空语句一般用在循环语句中，当需要构造空转循环体时，就是用空语句。

#### 6. 复合语句

我们已经知道了一个表达式后面加上一个分号就构成了一条语句。那么如下的写法就构造了三个语句:

```
a=1;  b=2;  c=3;
```

但是，在C++的某些特定的应用中规定了实现一个复杂的计算只能用一条语句来编写（例如: C++规定在循环语句和条件语句的判断条件之后，都只能跟一条语句作为循环体或者执行语句)。用一条简单的语句是无论如何都无法实现这些复杂计算的，这时就需要有一种方法将多条语句组合成一条语句，复合语句就是这样一种方法。

所谓复合语句就是用一对“{}”将若干条语句括在一起，系统会将这一组语句视为一条语句。将上面的三条语句修改如下:

```
{  a=1;  b=2;  c=3;  }
```

这时，原先分开的三条语句因为“{}”的作用，被视为一条语句，达到了我们想要实现的用一条语句实现复杂功能的要求。

## 4.3　选择结构语句的使用

在结构化程序设计的三种结构中，顺序结构就是依次编写语句，系统按照语句书写的先后次序，依次执行它们，以实现预先设计的功能。顺序结构不需要特别的语句来控制，因此不需要进行特别的介绍。

选择结构是三种基本结构中的第二种结构，它的主要功能是判断给定的条件，根据判断的结果从给定的两组操作中选择一组执行。

### 4.3.1　if语句

在C语言、C++等支持结构化程序设计的编程语言中，专门设计了一个条件语句来实现这个基本结构。条件语句也称为if语句，if语句最常见的形式如下:

```
if (<表达式>)  语句1
else           语句2
```

主要作用是: 当逻辑表达式的运算结果为真的时候，执行语句1; 否则，当逻辑表达式的运算结果为假的时候，就执行语句2。语句1和语句2存在一种互斥的关系，当程序运行时，只能在其中选择一条语句来执行。其操作流程见后面的图4-10。

为了适应程序设计中多种情况的处理，在C++中将if语句设计成了三种形式。

#### 1. 不平衡if语句

书写的格式如下:

```
if( <表达式> )  语句
```

流程图表示如图4-9所示。

应用实例如例4.1所示。

**例4.1**　演示不平衡if语句的使用方法。程序的功能是由用户输入三个数字，程序求出并输出其中的最大值。

```
#include <iostream.h>
void  main( )
{
     float a,b,c,t;
```

```
    cout << "请输入三个数:";
    cin  >> a >> b >> c;
    t=a;
    if(t < b)  t=b;
    if(t < c)  t=c;
    cout << "最大值:" << t << endl;
}
```

不平衡if语句主要用于简单的条件判断，决定某条语句或者某个有多条语句组成的功能是否执行。

### 2. 平衡的if语句

书写的格式如下:

```
if( <表达式> )  语句1
else            语句2
```

其操作流程见图4-10，其中<表达式>与图4-4中的逻辑条件C，“语句1”和“语句2”分别对应其中的A和B分支语句。应用实例如例4.2所示。

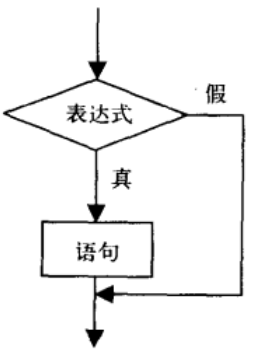



图4-9 不平衡的if语句

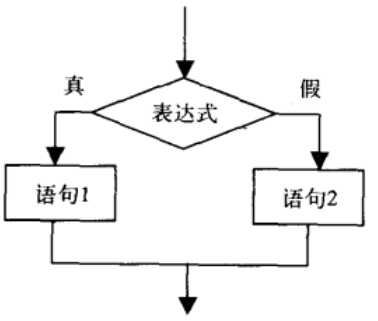



图4-10 平衡的if语句

**例4.2** 演示平衡if语句的使用方法，程序的功能与例4.1相同。

```
#include <iostream.h>
void  main( )
{
     float a,b,c,t;
     cout << "请输入三个数:";
     cin  >> a >> b >> c;
     if(a < b)  t=b;
     else       t=a;
     if(t < c) cout << "最大值:" << c << endl;
     else      cout << "最大值:" << t << endl;
}
```

在平衡的if语句中，程序员需要为用户提供两个选择语句或者功能，程序会根据用户输入的条件在两个选择语句或者功能中自动选择一个来执行。至少要执行一个选择分支语句或者功能。

### 3. 组合的if语句

上面介绍的两种if语句，只能够通过判断一个表达式来进行分支语句的选择。但是，在实际的程序编制过程中，有很多复杂的条件判断不是仅仅通过一次判断就可以实现的，这就需要有功能更强大的语句来完成复杂的判断工作。组合的if语句就是这样一个可以对多个逻辑表达式进行判断的功能强大的选择

语句。

书写的格式如下:

```
if( <表达式1> )          语句1
else if( <表达式2> )     语句2
else if( <表达式3> )     语句3

else if( <表达式n> )     语句n
else                     语句n+1
```

流程图表示如图4-11所示。

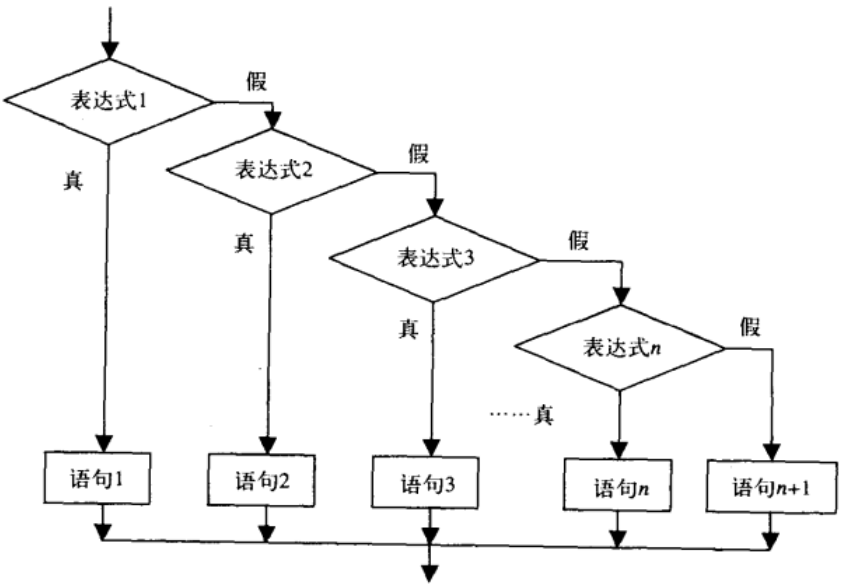



图4-11　组合的if语句

组合的if语句的应用实例如下面两例所示。

**例4.3**　演示组合的if语句的使用方法，程序的功能与例4.1相同。

```
#include <iostream.h>
void  main( )
{
     float a,b,c;
     cout << "请输入三个数:";
     cin  >> a >> b >> c;
     if(a>=b && a>=c)  cout << "最大值:" << a << endl;
     else  if(b>=a && b>=c)  cout << "最大值:" << b << endl;
     else  cout << "最大值:" << c << endl;
}
```

**例4.4**　演示组合的if语句的使用方法，程序的功能为用户输入百分制成绩，程序将其转换为五分制成绩输出。

```
#include <iostream.h>
void  main( )
{
   float score;
```

```
    cout << "请输入成绩:";
    cin  >> score;
    if(score>=90)  cout << "成绩等级: A" << endl;
    else  if(score>=80)  cout << "成绩等级: B" << endl;
    else  if(score>=70)  cout << "成绩等级: C" << endl;
    else  if(score>=60)  cout << "成绩等级: D" << endl;
    else  cout << "成绩等级: E" << endl;
}
```

从上面两例中可以看出，组合的if语句书写较为复杂，往往一条语句会占据多行空间。因此，在使用组合的if语句时，必须要注意无论书写了多少行代码，它都是一条语句。同时还要注意，在组合的if语句中，每一条else语句都包含了逻辑条件，如果能够很好地理解else语句中所包含的逻辑条件就可以大大简化后续if语句的书写复杂度。

上述是if语句的三种表示形式（注意：三种格式中的“语句1”、“语句2”、……、“语句*n*”都必须为一条语句，可以是简单语句，也可以是复合语句）。虽然它们的功能已经足够强大了，但是在实际使用时，如果单独使用一条if语句往往不能完全实现复杂的逻辑。因此，在C++中除了允许使用单条if语句以外，还允许将if语句嵌套使用。

### 4.3.2 if语句的嵌套使用

在if语句中又包含了一个或多个if语句，这称为if语句的嵌套。一般形式如下：

```
if(  )
   if(  )    语句1   } 内嵌if
   else      语句2   }
else
   if(  )    语句3   } 内嵌if
   else      语句4   }
```

应当注意if与else的配对关系。根据C++的语法规定，else总是与写在它前面的、最靠近的、尚未与其他else配对过的if配对。假如写成：

```
if(  )
   if(  )    语句1      //内嵌if
else
   if(  )    语句2   }
   else      语句3   }   内嵌if
```

程序员把第一个else写在与第一个if（外层if）同样突出的位置上，本意是希望这个else能够与第一个if对应。但根据上面所说的语法规定，由于第二个if没有与else配过对，又最靠近第一个else，因此实际上第一个else是与第二个if配对。这样，在程序的实际执行过程中就得不到我们需要的结果。因此最好使内嵌的if语句也含有else部分，这样if的数目和else的数目相同，从内层到外层一一对应，不致出错。

如果if与else的数目不一样，为了达到程序员的设计目的，不至于造成误解，应该加入“{}”符号，将内嵌的不平衡if语句构造成一条复合语句。这样，else不能和复合语句中的if配对，避免了由于配对错误造成的逻辑错误。可将上例改为：

```
if(  )
  {if(  )  语句1}       //内嵌if
else
if(  )   语句2    }
  else    语句3   }      内嵌if
```

这时由于“{}”限定了内嵌if语句的范围，因此else与第一个if配对，达到了我们预期的要求。

下面再举一个实际的数学计算题来帮助理解if语句在程序中的应用方法。有一个数学函数如下：

$$y=\begin{cases}-10 & (x<10)\\ 5 & (x=10)\\ 20 & (x>10)\end{cases}$$

使用if语句解题的实际应用如例4.5所示。

**例4.5**　演示用嵌套的if语句解决数学问题。

```
#include <iostream.h>
void  main( )
{
     float x,y;
     cout << "请输入x变量的值: ";
     cin  >> x;
     if(x < 10)  y = -10;
     else  if(x == 10)  y = 5;
          else  y = 20;
     cout << "y= " << y << endl;
}
```

相关的流程图，如图4-12所示。

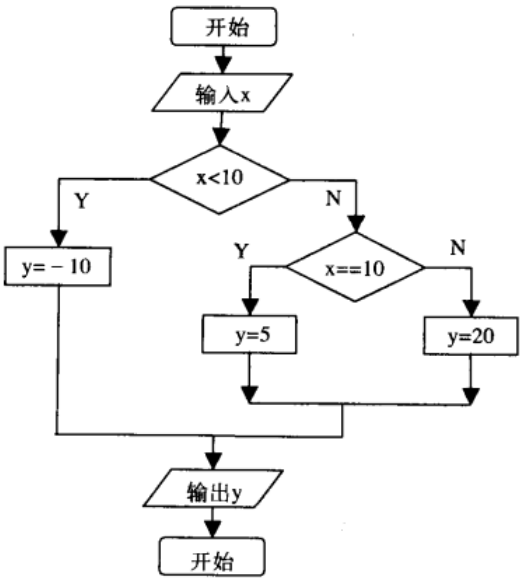



图4-12　用嵌套的if语句解决数学问题

### 4.3.3　条件运算符

在C语言和C++中，针对简单的平衡的if语句，还有另外一种表示方式，即条件运算符。用条件运算符构成的表达式称为条件表达式，条件表达式的一般形式书写如下：

```
<表达式1> ? <表达式2> : <表达式3>
```

条件表达式的执行流程如图4-13所示。

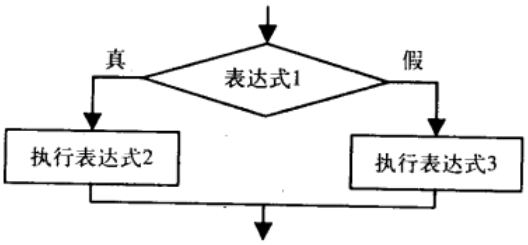



图4-13　条件运算符

条件运算符是一种三元运算符，即在表达式中需要书写三个操作数。根据图4-13所描述的流程图可以看出，条件运算符的使用方法与平衡的if语句完全相同。首先计算表达式1（一般为逻辑表达式）的值，如果表达式1的值为真（值为非0）则执行表达式2并将其计算结果作为整个表达式的值；如果表达式1的值为假（值为0）则执行表达式3并将其计算结果作为整个表达式的值。表达式1对应的是平衡的if语句中的判断表达式，表达式2对应的是平衡的if语句中的真分支语句，表达式3对应的是平衡的if语句中的假分支语句。

例如在求两个数中最大数的计算过程中，可以使用条件运算符书写如下：

```
max = (a>b) ? a : b;
```

也可以使用if语句书写如下：

```
if(a>b)  max = a;
else     max = b;
```

当然由于在if语句中可以使用复合语句来实现复杂的功能，所以条件运算符的功能只能相当于简单的平衡的if语句，而不能认为两者是完全等价的。

条件运算符的计算优先级比较低，仅仅高于各种赋值运算符和逗号运算符。因此在条件运算表达式中，一般是将所有的逻辑运算和数值运算都计算完毕以后，使用条件运算符选择合适的分支取得计算结果，最终再将计算结果赋值给其他变量。

条件运算符是按照“自右向左”的计算规则进行计算的。当嵌套使用条件运算符时，这个计算规则

将起到重要的作用。例如有如下的条件运算符表达式:

```
x = a>b ? a : c<b ? c : b;
```

在进行计算时的实际执行规则如下:

```
x = a>b ? a : (c<b ? c : b);
```

先计算右侧的条件运算符，然后将计算结果再次带入表达式，继续计算左侧的条件运算符。若a=1，b=3，c=5，则x的值为3。

再例如需要求解三个数中的最大数时，也可以使用条件运算符将程序书写如下:

```
max = (a>b ? a:b)<c ? c : a>b ? a:b;
```

从上面的例子可以看出，虽然可以利用“自右向左”的规则来实现正确的计算，但是这样的书写方法不利于程序的阅读和维护。所以，在嵌套使用条件运算符的时候，应该尽量使用括号将条件运算符及相关数据括起来，防止出现执行和理解错误。所以上面求解最大数的例子应该写为:

```
max = (a>b ? a:b)<c ? c : (a>b ? a:b);
```

### 4.3.4 开关语句

根据前面的介绍，可以使用组合的if语句来实现复杂的逻辑判断。但是我们同时也提到，组合的if语句书写工作量大，使用不太方便。因此，在C++中也设计了一个功能类似而使用简单的语句: 开关语句（switch语句）。

开关语句也称为多分支选择语句，功能与组合的if语句类似，可以用来模拟组合的if语句的功能（两者的功能并非完全等价）。开关语句通常用于各种分类统计和计算方面，对多个按照某种条件进行的分类进行选择。该语句的一般表示形式如下:

```
switch( <表达式> )
{
  case <常量表达式1>: 语句序列1;
                      [< break; >]
case <常量表达式2>: 语句序列2;
                      [< break; >]
......
case <常量表达式n>: 语句序列n;
                      [< break; >]
  default:             语句序列n+1;
}
```

该语句在每个分支语句的后面加上break语句后的操作流程图如图4-14所示。

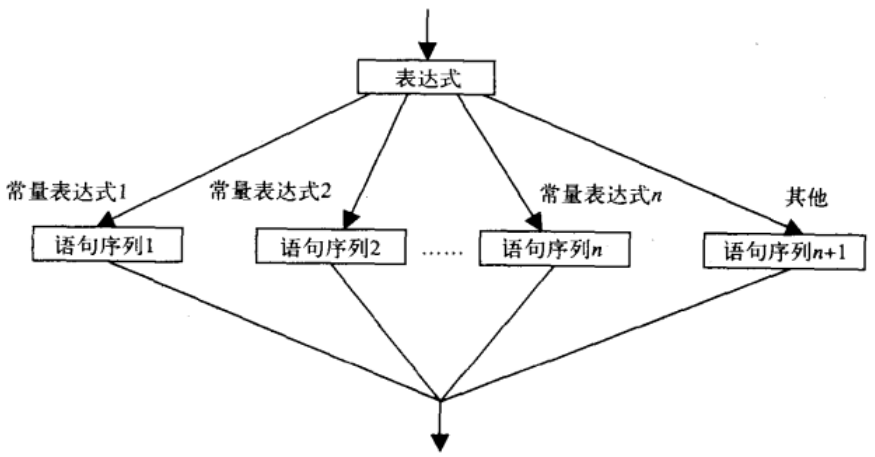



图4-14 开关语句流程

从图中可以看出switch语句的执行流程。首先计算关键字switch后面括号中的“表达式”的值。注意，表达式的计算结果只能是字符类型或者整数类型。然后用表达式的值和下面多个关键字case后面的常量表达式的值对比，好像是用表达式的值查询一个常量列表。当然这里每个常量表达式的值也必须是字符类型或者整数类型的常量。如果在下面的常量表达式列表中找到了一个与表达式的值完全匹配的分支，则执行这个常量表达式后面跟随的语句序列。如果在所有的常量表达式中都没有找到匹配的数值，则执行关键字default后面跟随的语句序列。

例如实现一个程序，由用户输入英文五分制的成绩，程序按照中文五分制的格式输出成绩。程序的主要分支处理语句如下:

```
switch(x)
{   case 'A': cout << "优" <<endl;
    case 'B': cout << "良" <<endl;
    case 'C': cout << "中" <<endl;
    case 'D': cout << "及格" <<endl;
    default:  cout << "不及格! " <<endl;
}
```

在switch语句中，每个常量表达式后面跟随的语句序列可以是一条语句，也可以是多条语句。相当于在组合的if语句中的每个分支中都可以使用简单语句和复合语句的使用方法。但是需要注意的是，虽然每个常量表达式后面的语句序列类似于if语句的分支，但是它们的执行方法并不完全相同。考虑如下的情况:

```
int x;
...
if(x>=-2 && x<0)        cout << "负数";
else if(x==0)           cout << "零" ;
else if(x>0 && x<=2)    cout << "正数" ;
```

在if语句中使用了大于号和小于号等符号来表示一个数值的范围，这在if语句中是很常见的操作。而由于switch语句使用的是匹配原则，相当于在if语句中使用等于操作符，这样就很难进行一个范围的数值描述。为了使switch语句也能像if语句一样描述一个范围的数据条件，在C语言和C++中对其做了特殊的规定。

switch语句规定，当找到一个匹配的常量表达式并且执行了后面的语句序列后，并不像if语句那样结束当前语句的执行，而是继续执行下一个常量表达式后面的语句序列，直到执行完全部的语句序列或者遇到break语句时终止。这样就可以使多个数值共享一个语句序列，这在一定程度上实现了描述一个数值范围的操作。例如可以将上面由if语句实现的程序段改写成如下程序段:

```
int x;
...
switch(x)
{ case  -2:
  case  -1: cout << "负数";
            break;
  case   0: cout << "零" ;
            break;
  case   1:
  case   2: cout << "正数";
}
```

在程序段中，如果x的值为-2，则与第一个case语句匹配。由于第一个case语句后面没有执行语句序列，也没有break语句，因此根据规则，程序会继续执行第二个case语句后面的语句序列。数值-1和-2共享了输出“负数”字符串的语句序列，而且紧跟着写有break语句，程序在执行了输出语句后退出

switch语句，这样实现的功能与上述if语句程序段完全相同。

注意:

（1）在switch语句中，default分支可以放在任何位置。但是为了配合常量表达式匹配查询的执行流程，使得程序书写和阅读方便，一般将default分支写在switch语句的最后一行。

（2）每个常量表达式的值都必须互不相同，否则当表达式的值与多个常量表达式的值都匹配时，计算机将无法决定到底执行哪一个常量表达式后面的语句序列。

（3）常量表达式必须是一个确定的字符类型或者整数类型的常量数值，而不能是浮点数或者变量表达式。

（4）switch语句虽然可以模拟组合的if语句，但是并不是与组合的if语句完全等价。switch语句只能处理字符类型和整数类型的条件判断，而组合的if语句不但可以处理字符类型和整数类型的条件判断，还可以对浮点数等其他的条件进行判断。由此看来，组合的if语句可以实现的功能要远远大于switch语句。

下面具体举例说明switch语句在实际生活中的应用。

在单位的工资发放中都要缴纳住房公积金，住房公积金是根据收入的多少按不同比例缴纳的，假设比例设置如下:

```
          收入 < 1000          2%比例
 1000 <= 收入 < 2000          3%比例
 2000 <= 收入 < 5000          4%比例
 5000 <= 收入 < 10000         5%比例
10000 <= 收入                 6%比例
```

实际应用程序如例4.6所示。

**例4.6** 演示用switch语句解决公积金问题。程序的功能是根据输入的收入金额求出并且输出应该缴纳的公积金。

```
#include <iostream.h>
void  main( )
{
     int  in,temp,r;
     float  fee;
     cout << "请输入您的收入: ";
     cin  >> in;
     temp = in/1000;      //按照1000为单位进行区间划分
     switch(temp)
     {  case 0: r=2; break;
        case 1: r=3; break;
        case 2:
        case 3:
        case 4: r=4; break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9: r=5; break;
        default: r=6;
     }
     fee = in * r / 100.0;
     cout << "公积金为: " << fee << endl;
}
```

程序使用了多个分支共享语句序列的技术，使得switch语句能够处理一个范围内的数据。在switch语

句的最后使用了default语句，用来处理收入>=10000后的所有情况。

## 4.4　循环结构语句的使用

计算机区别于其他计算工具的主要特点之一就是计算机能够存储和自动运行程序。现代计算机是一个可以实现高速运算的机器，每秒钟可以进行几亿次计算，如果是一个顺序执行的程序，就算是程序员不停编写几个月甚至几年的程序，也可以在瞬间执行完毕。而我们却经常能够看到一些大型的科学计算需要计算机不停地计算几天、几个星期，甚至几个月。这种长时间自动运行程序的功能不是靠程序员编写很多代码来实现的，而主要是靠循环结构使得程序反复迭代计算实现的。

只要计算的数据之间存在着规律性的变化，就可以使用循环结构来对其进行迭代计算。例如使用循环结构可以实现多项式累加和与累乘积的求解:

$$\sum_{i=1}^{n} i = 1+2+3+\cdots+n$$

$$n! = 1\times 2\times 3\times 4\times\cdots\times n$$

$$\pi = 4\times\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots+(-1)^{n+1}\frac{1}{2n-1}+\cdots\right)$$

$$\cos(x) = 1-\frac{x^2}{2!}+\frac{x^4}{4!}-\cdots+(-1)^{n+1}\frac{x^{2n-2}}{(2n-2)!}+\cdots$$

循环结构的分类和运行原理在4.1.4节中已经进行了简单的介绍，其在C++中的实现形式主要有以下四种:

（1）用goto语句和标号构成循环。

（2）用while语句构成循环。

（3）用for语句构成循环。

（4）用do-while语句构成循环。

下面我们分小节来分别进行介绍。

### 4.4.1　goto语句及标号的使用

goto语句最早出现在顺序结构的程序设计语言中，负责程序模块之间的跳转工作。它的一般书写形式为:

```
goto  <语句标号>;
```

<语句标号>是一个标识符，它被用来标记程序跳转的位置。下面使用goto语句编写程序例4.7来求解 $\sum_{i=1}^{100} i$ 。

**例4.7**　演示用goto语句求解 $\sum_{i=1}^{100} i$ 。

```
#include <iostream.h>
void  main( )
{
  int  i=1,m=0;
  loop:  if(i <= 100)
           { m += i;
             i++;
             goto  loop;     //跳转到标号loop
           }
  cout << "从1加到100的总和是: " << m << endl;
}
```

由于发现在程序中大量使用goto跳转命令会造成程序结构的混乱，而且虽然goto语句和if语句联用可

以实现循环结构，但是在C++中我们有能够实现相同功能的、结构更加简单的语句；因此，该命令使用得越来越少了，甚至程序员往往会被建议不要在程序中使用该命令。

### 4.4.2 while语句

while语句与其字面上的含义一致，是一种典型的“当型”循环结构。其一般书写形式如下：

```
while(<表达式>)  <语句>
```

执行的顺序为：首先判断表达式的计算结果，如果计算结果为真（非0值）则执行while语句后面跟随的语句序列，直到表达式的计算结果为假（0值）的时候，结束循环语句的执行。我们一般将while语句后面跟随的语句序列称为循环体，它是在判断表达式控制下进行运行的一组整体代码。对应的流程图如图4-15所示。

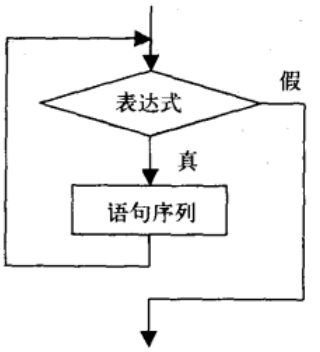


图4-15 while语句流程

下面使用while语句编写程序例4.8来求解 $\sum_{i=1}^{100} i$ 。

**例4.8** 演示用while语句求解 $\sum_{i=1}^{100} i$ 。

```
#include <iostream.h>
void  main( )
{
   int  i=1,sum=0;
   while(i <= 100)
   {  sum += i;
      i++;
   }
   cout << "从1加到100的总和是: " << sum << endl;
}
```

在上述的程序中需要注意while语句的几个用法：

（1）与if语句相同，while语句后面也只能跟随一条语句。当用户想要在循环体内实现较为复杂的功能时，必须要将多条语句用“{}”符号括起来，构成一条复合语句。这样才能正确地实现程序的功能。后面介绍的for语句和do-while语句与while语句相同，其后都只能跟随一条语句，都只能使用复合语句来实现复杂的功能。

（2）在while语句的循环体内必须要包含对循环判断表达式中循环变量修改的语句。例如，在程序例4.8中结束循环的条件是变量i的值大于100。由于变量i的值决定了是否结束循环，是对循环起到至关重要作用的变量，因此我们将其称为循环变量。还由于变量i初始的值为1，如果在循环体内没有语句i++对其值进行修改，则变量i的值永远为1，循环永远无法结束，我们称其为死循环。死循环会导致程序死锁或者计算机死机，这是绝对不允许出现的，因此在循环体内必须要包含使得循环趋向于结束的循环变量修改语句。

（3）常见的循环操作有两类：累加计算和累乘计算。针对不同的循环操作，需要设置不同的初始化值。累加计算中的累加和变量一般初始化为0，累乘计算中的累乘积变量一般初始化为1。

### 4.4.3 for语句

for语句与while语句功能相同，也是一种典型的“当型”循环结构。其一般书写形式如下：

```
for([<初始化表达式>]; [<条件表达式>]; [<修正表达式>])
      <语句>
```

执行的顺序为：首先执行<初始化表达式>完成某些循环语句中用到的变量的初始化。然后进行<条

件表达式>的计算，这里条件表达式的功能等价于while语句中的判断表达式。如果条件表达式的计算结果为真（非0值），则执行for语句后面跟随的语句，然后执行<修正表达式>，修改相关的循环变量，继续下一次循环；如果<条件表达式>的计算结果为假（0值），则结束循环语句的执行。

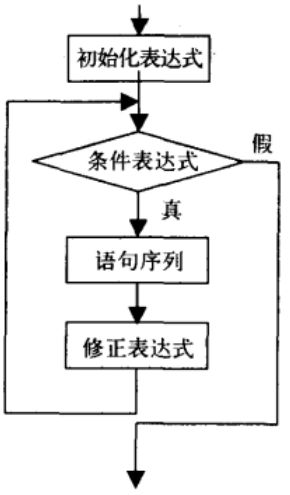


图4-16　for语句流程

在for语句中我们同样将后面跟随的语句序列称为循环体。与while语句不同的是，在for语句中一般将对循环变量修改的操作从循环体中分离出来，单独作为修正表达式来书写。对应的流程图如图4-16所示。

从流程图的结构可以看出，while语句与for语句都属于“当型”循环，只不过for语句将原来while语句之前独立的初始化赋值语句合并到语句中成为初始化表达式，将原来循环体中循环变量的修改操作独立出来形成修正表达式而已。如果将初始化表达式取消，将循环体语句序列和修正表达式合并，则两种循环的流程图将完全一致。下面使用for语句编写程序例4.9来求解 $\sum_{i=1}^{100} i$ 。

例4.9　演示用for语句求解 $\sum_{i=1}^{100} i$ 。

```
#include <iostream.h>
void  main( )
{
   int  sum=0;
   for(int i=1; i<=100; i++)
     sum += i;
   cout << "从1加到100的总和是: " << sum << endl;
}
```

根据程序例4.9我们可以看出，用for语句实现与while语句同样的循环功能时，程序可以书写得更加简单。由于for语句能够实现与while语句完全相同的功能，所以一般程序员在编写程序时有使用for语句取代while语句的趋势。for语句功能十分强大，使用for语句除了标准的用法外，还有一些特殊的用法和注意事项，如下所示：

（1）在C++中变量的定义可以在任何地方实现。因此，在C++的for语句的初始化表达式中，不但可以为变量赋初始值，还可以定义变量。例如：

```
for(int i=0; i<100; i++);
```

（2）在标准的写法中，for语句的三个表达式都要书写。但是在实际使用时可以针对不同情况，省略一个或者多个表达式，以达到特殊的使用效果。

如果将初始化的语句放到for语句的前面书写，则可以在for语句中省略初始化表达式。例如：

```
int i=0;
for(; i<100; i++);
```

在for语句中也可以省略条件表达式，这时候的for语句的功能相当于while语句的一种特殊写法，如下所示：

```
for(int i=0;  ; i++)     ⇔      while(1)
```

这种循环我们称为永真循环，即永远循环或者死循环。这种循环配合我们在4.4.5节中介绍的continue和break语句，可以实现一些具有特殊功能的循环计算。

当我们将循环变量的修改语句写入循环体中时，for语句中的修正表达式也可以省略。例如:

```
for(int i=0; i<100; )
{  cout << i << endl;
   i++;
}
```

有时也可以将三个表达式都省略，这时for语句同样等价于永真循环，如下所示:

```
for( ; ; )      ⇔      while(1)
```

(3) for语句和while语句都是“当型”循环，它们的执行方法是先执行条件语句，再根据条件语句的计算结果判断是否执行循环体中的语句。由于是先判断后执行，因此循环体可能一次都不会被执行。

### 4.4.4 do-while语句

do-while语句虽然字面上与while语句都含有一个while单词，但是它们的执行流程正好相反。do-while语句先执行循环体，然后再判断是否要继续执行循环，属于典型的“直到型”循环。其一般书写形式如下:

```
do
  <语句>    //循环体
while(<表达式>);
```

执行的顺序为: 首先执行循环体中的语句序列（其中也涉及循环变量的修改），在循环体执行完毕以后再判断表达式的计算结果。如果计算结果为真（非0值）则进行下一次的循环体运算，直到表达式的计算结果为假（0值）的时候，才结束循环语句的执行。只要保证在循环体内对循环变量的修改是趋向于使得循环结束的，就可以使得do-while循环正常结束。对应的流程图如图4-17所示。

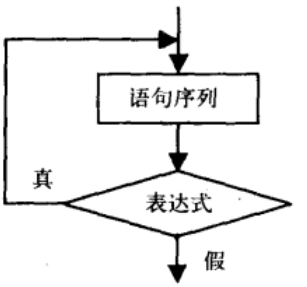


图4-17 do-while语句

下面使用do-while语句编写程序例4.10来求解 $\sum_{i=1}^{100} i$ 。

例4.10 演示用do-while语句求解 $\sum_{i=1}^{100} i$ 。

```
#include <iostream.h>
void  main( )
{
   int  i=1,sum=0;
   do {
     sum += i;
     i++;
   }
   while(i <= 100);
   cout << "从1加到100的总和是: " << sum << endl;
}
```

由于do-while循环是“直到型”循环，先执行循环体后判断。所以与“当型”循环的循环体有可能一次都不被执行不同，其循环体至少会被执行一次。这种差别会导致同样书写的代码，由于采用的循环语句不同而计算的结果不同。

下面使用while语句和do-while语句来实现相同的计算数值累加和的算法，以比较两种循环的不同点。

```
void  main( )
{
  int i,sum=0;
  cin >> i;
```

```
void  main( )
{
  int i,sum=0;
  cin >> i;
```

```
    while(i <= 5)
    {
       sum += i;
       i++;
    }
    cout << sum << endl;
}
```

```
do
 {  sum += i;
    i++;
 }
 while(i <= 5);
 cout << sum << endl;
}
```

在上述的程序中，如果用户输入的数据小于或者等于5，则两个程序的功能等效，都是求解从输入的数据到5的累加和。如果输入的数据大于5，则两个程序的功能不等效。其中使用while语句的程序由于先判断再计算，导致了由于判断条件为假而不做计算的结果，输出的结果始终为0。而使用do-while语句的程序由于先执行后判断，会对用户输入的数据至少计算一次。例如用户输入6，则使用do-while语句的程序输出6，而使用while语句的程序则输出0。

### 4.4.5 break语句和continue语句

在循环语句的执行过程中，主要是按照循环语句的条件表达式的计算结果来决定何时结束循环语句的执行。如果在程序的循环中要针对某些特殊情况做出特别的处理，例如强行中断循环的执行，跳过某些特定的执行过程等，这就需要加入新的控制语句了。break语句和continue语句正是专门设计的用来改变循环的执行流程的语句。

我们在前面switch语句的使用介绍中已经说明了：break语句可以中断switch语句中分支语句顺序向下执行的流程。不过，break语句不是仅仅用在switch语句中，它更重要的作用是用来中断循环语句的执行。

break语句的书写格式一般为：

```
break;
```

其功能是：中断程序的执行流程。下面用例子说明：

```
for(int i=1;  ; i++)
{   cout << i << endl;
    if(i >= 10) break;
}
```

程序段的作用是在屏幕上显示1～10十个数。在for语句的书写上使用了永真循环的写法，程序本来应该永远不停地循环下去。但是由于循环体内使用了break语句，当变量i的值等于10的时候，if语句中的break语句被执行，程序的执行流程跳出循环语句，结束本来应该无限循环的循环语句。

break语句可以用来打断循环语句的执行流程，跳出循环语句去执行紧跟在循环语句后面的下一条语句。但是必须注意，一条break语句只能跳出它所在的最内层循环，如果将break语句写在一个4.4.6节介绍的多重循环语句中的时候，想要跳出循环语句往往需要多次调用break语句。

continue语句也是一种改变循环执行流程的语句，但是它的作用与break语句不同。它并不使程序的执行流程跳出循环语句，而是跳过循环体中下面尚未执行的语句，根据条件表达式的计算结果决定是否要执行下一次循环。

continue语句的书写格式一般为：

```
continue;
```

我们主要使用continue语句跳过循环过程中某些特别的计算过程。必须要注意，continue语句一定要配合if语句写在循环体的中间，将需要跳过的计算过程写在continue语句的后面，这样才能达到预期的效果。

下面使用break语句编写程序例4.11来求解某个用户输入的数据是否是素数。

**例4.11**　演示用break语句求解素数。程序的功能是由用户输入一个整数，经过运算后判断其是否是一个素数。

```
#include <iostream.h>
#include <math.h>
```

```
void  main( )
 {
    int a,b,i;
    cout << "请输入一个整数: ";
    cin >> a;
    b = sqrt(a);
    for(i=2;i<=b;i++)
      if(a%i==0) break;
    if(i>=b+1)
      cout << a << "是素数" << endl;
    else
        cout << a << "不是素数" << endl;
  }
```

break语句和continue语句的对比实例如下:

```
for(i=1; i<=4; i++)                    for(i=1; i<=4; i++)
{   x = i * i;                         {     x = i * i;
    if(x == 9) break;                        if(x == 9) continue;
    cout << x << endl;                       cout << x << endl;
}                                      }
cout <<"i="<< i << endl;               cout <<"i="<< i << endl;
```

上面的两个程序段中除了break语句和continue语句不同之外，其他都完全相同，但是它们的执行结果却完全不同。结果如下所示:

```
1              1
4              4
i=3            16
               i=5
```

当循环变量i的值为3时，在循环体内会执行break语句或者continue语句。如果执行break语句，则退出循环，这时后续的结果都不会显示出来。如果执行continue语句，则跳过显示9的语句，将循环执行完毕。

### 4.4.6 循环的嵌套

在某些复杂计算中，使用一个循环语句无法实现计算的目标，这时往往需要在一个循环语句中再使用循环语句。这种在一个循环语句中又包含其他循环语句的循环结构称为循环的嵌套。在前面所介绍的三种主要的循环语句（while语句、for语句和do-while语句）中，彼此之间都可以互相嵌套。

我们将循环嵌套调用的包含关系称为层次关系。一个循环语句包含另外一个循环语句称为两层循环。在C++中，对循环嵌套调用的层次是无限的，在复杂的问题求解中甚至会用到4层或5层循环。当然，在循环的嵌套调用中也不仅仅存在层次关系，还存在并列关系。

下面以图示说明几种常见的循环嵌套结构，图4-18中每一个闭环矩形代表一个循环语句。

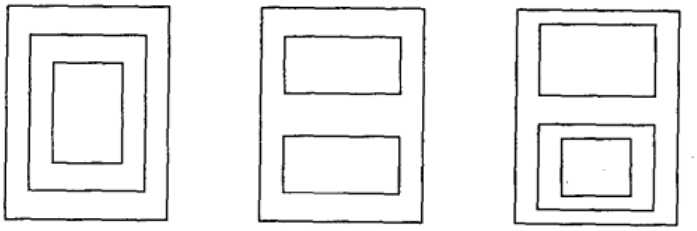



图4-18 循环嵌套类型

下面使用循环嵌套技术编写程序例4.12来输出99乘法表。

**例4.12**　演示用循环嵌套技术输出99乘法表。

```
#include <iostream.h>
void  main( )
{
  int i,j;
  cout << "乘法表: " << endl;
  for(i=1;i<=9;i++)          //外循环控制输出的行数
  {
      for(j=1;j<=i;j++)      //内循环
        cout << i << "×" << j << "=" << i*j << '\t';
      cout << endl;
  }
}
```

注意：由于for语句中的条件判断部分一般要明确写入循环的次数，因此一般用for语句来处理预知次数的循环操作，而while语句主要用来处理未知循环次数的循环操作。

## 4.5　控制语句的应用举例

**例4.13**　求解一元二次方程$ax^2+bx+c=0$的根。

在求解的过程中有几种情况需要考虑：

(1) $a=0$，则表达式不是一元二次方程，解为$-\frac{c}{b}$。

(2) $b^2-4ac=0$，为两个相等的实数根。

(3) $b^2-4ac>0$，为两个不相等的实数根。

(4) $b^2-4ac<0$，为两个共轭的复数根。

根据以上分析，我们对各种情况使用if语句来分别处理，编写程序如下：

```
#include  <iostream.h>
#include  <math.h>
void main( )
{
      float a, b, c, disc, x1, x2, real, image;
      cout << "请输入三个实数: ";
      cin >> a >> b >> c;
      cout << "\n方程";
      if (fabs(a)<=1e-7)              //a==0, 注意写法
          cout << "不是二次方程, 解为: " << -c/b << endl;
      else {                          //处理其他三种情况
         disc=b*b-4*a*c;
         if (fabs(disc)<=1e-7)        //disc==0
             cout << "有两个相等的实数根: " << -b/(2*a) << endl;
         else if (disc>1e-7)
         {  x1=(-b+sqrt(disc))/(2*a);
            x2=(-b-sqrt(disc))/(2*a);
            cout << "有两个不相等的实数根: " << x1 << " 和 " << x2 <<endl;
         }
         else
         {  real=-b/(2*a);
            image=sqrt(-disc)/(2*a);
```

```
            cout << "有两个复数根: " << endl;
            cout << real << "+" << image << "i" << endl;
            cout << real << "-" << image << "i" << endl;
        }
    }
}
```

由于程序中$b^2-4ac$是一个必须计算的部分，我们用一个中间变量disc来存放这部分值，以避免重复计算。由于我们设定disc为实数变量，因此在计算机中实数变量存放的值并非绝对精确。因此判断disc是否为0的时候，我们不是绝对取0值，而是取一个接近0的区间（小于$10^{-7}$）。程序的运行结果如下：

（1）请输入三个实数: 0 1 2

方程不是二次方程，解为: -2

（2）请输入三个实数: 1 4 4

方程有两个相等的实数根: -2

（3）请输入三个实数: 2 4 1

方程有两个不相等的实数根: -0.292893 和 -1.70711

（4）请输入三个实数: 2 2 1

方程有两个复数根:
-0.5+0.5i
-0.5-0.5i

在4.4节中我们举例说明了循环语句能够实现的有规律的科学计算。我们可以将其分为两类：有穷计算和无穷计算。

对于有穷计算，由于已经知道了计算的步数，最适合用for语句来实现。对于无穷计算，当然不可能无止境地计算下去，我们一般使用while语句或者do-while语句，通过在条件表达式中对计算公式通项的判断来决定何时结束循环。能够求得结果的多项展开式，其通项都是收敛的（越变越小的）。我们认为当通项足够小的时候（例如，小于$10^{-7}$），就可以对以后的计算忽略不计，以后的计算也不会对解的大小产生足够的影响。因此，对于无穷计算往往用while语句或者do-while语句判断通项是否足够小来决定循环是否继续计算。

for语句除了可以处理预知次数的循环操作，也可以通过对条件表达式的判断来处理未知循环次数的无穷计算。因此，for语句的使用频率要高于while语句和do-while语句。

下面我们通过实际程序的编写来说明循环语句的使用方法。

**例4.14** 使用循环语句求解cos(*x*)的多项式求和问题。

cos(*x*)的多项式求和公式如下所示：

$$\cos(x)=1-\frac{x^2}{2!}+\frac{x^4}{4!}-\cdots+(-1)^{n+1}\frac{x^{2n-2}}{(2n-2)!}+\cdots$$

程序编写如下：

```
#include  <iostream.h>
#include  <math.h>
void main( )
{
  int i;
  float x,t,value;
  cout << "请输入x的值: ";
  cin >> x;
  value=1; t=1; i=1;
```

```
    while((fabs(t))>=1e-9)
    {
        t=t*(-1)*x*x/((2*i)*(2*i-1));        //通项的值
        value = value + t;
        i++;
    }
    cout << "cos(x)=" << value << endl;
}
```

由于求解cos($x$)的多项式的项数是无穷的，所以我们使用while语句，通过判断通项值的大小来结束循环。当通项的值小于$10^{-9}$时，我们认为这以后的值对计算结果的影响不大，可以忽略不计，于是结束循环。

程序中我们使用变量n来记录分子的变化，变量m记录分母的变化，变量t记录通项的值，变量sign记录符号的变化。经过若干次累加，最后在变量cos中获得计算结果。

**例4.15**　使用循环语句求解 $\sum_{n=1}^{20} n!$ 的多项式求和问题（即求解1!+2!+…+20!）。

程序编写如下：

```
#include  <iostream.h>
void main( )
{
     double sum,t;
     sum=0;        t=1;
     for(int i=1; i<=20; i++)
     {
       t=t*i;        //计算每一个阶乘
       sum=sum+t;
     }
     cout << "sum=" << sum << endl;
}
```

由于本例中循环的次数确定，因此我们使用for语句来完成循环的工作。其中变量sum是用来存放累加和的，因此初始化成0；而变量t是用来存放累乘积的，因此初始化成1。

**例4.16**　使用循环语句求解Fibonacci数列的前20项。

该数列的运算规则如下：（1）第一数和第二个数都为1。

（2）从第三个数开始，每个数都是前两个数之和。

归纳成数学公式如下：

$$\begin{cases} F_1 = 1 \\ F_2 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

程序编写如下：

```
#include  <iostream.h>
void main( )
{
     int f1, f2;
     f1=1;   f2=1;
     for(int i=1; i<=10; i++)
     {
       cout << f1 << " " << f2 << endl;
```

```
        f1=f1+f2;
        f2=f2+f1;
    }
}
```

**例4.17** 使用循环语句求解100~200之间的所有素数。

首先，素数的概念是只能被1和其自身整除的数，因此我们可以使用试探法来寻找素数。假设我们要试探整数$x$是否是素数，就可以将1到$x$之间的所有整数都与$x$相除，如果没有一个数可以将$x$除尽，则$x$就是素数。又由于寻找素数其实是寻找一种倍数关系，所以没有必要扫描1到$x$之间的所有整数，只要扫描2到$\sqrt{x}$之间的整数就可以了。$\sqrt{x}$+1到$x$之间的整数是2到$\sqrt{x}$之间整数的倍数，只要前面区间的整数不能被整除，后面区间的整数也不会被整除。

程序编写如下：

```
#include  <iostream.h>
#include  <math.h>
void main( )
{
     int  m, k, i, n=0;                    // n是计数器
     for (m=101; m<=200; m=m+2)            // 2以外的偶数不会是素数
     {
          k=sqrt(m);
          for (i=2; i<=k; i++)
               if (m%i==0) break;
          if (i>=k+1)                      // 正常结束，是素数
          {  cout << m << "     ";
             n=n+1;
              if (n%10==0)                 // 控制每行输出10个素数
                  cout << endl;
          }
     }
}
```

本例中使用了双重嵌套循环技术，外层的循环是为了获得100到200之间的各个整数的值，内层的循环是为了探测当前整数是否是素数。如果内层的循环发现一个可以将当前整数除尽的数，则执行break语句终止内层循环的继续执行。在内层循环之后，会有一个判断语句来判断内层循环是否正常终止。如果是正常终止，则变量i的值会等于k+1，这时就将当前整数作为素数输出。否则，变量i的值会小于k+1，说明当前整数发现一个可以除尽的除数，它就不是素数。

**例4.18** 使用循环语句按照定义求任意两个数的最大公约数和最小公倍数。

程序编写如下：

```
#include  <iostream.h>
void main( )
{
  int  x, y, i, k;
  cout << "请输入两个整数: ";
  cin >> x >> y;
  k=x<y?x:y;                    // 取两数中的小数
  for(i=k; i>=1; i--)           // 求最大公约数
   if(x%i==0 && y%i==0)
   {
        cout << "最大公约数是: " << i << endl;
```

```
            break;
        }
    k=x>y?x:y;                          // 取两数中的大数
    for(i=k; i<=x*y; i++)               // 求最小公倍数
     if(i%x==0 && i%y==0)
     {
            cout << "最小公倍数是: " << i << endl;
            break;
     }
}
```

我们使用探测的方法，求最大公约数的时候就从两数中的小数开始，依次递减，直到1为止。这其间如果找到一个能够同时被两数整除的数，该数就是最大公约数，否则1是所有整数的公约数。同样的道理，从两数中的大数开始，依次递增，直到两数的乘积为止。无论寻找的过程中是否发现一个数同时是两个数的倍数，两数的乘积必定会是两个数的公倍数。

**例4.19**　求出并输出满足如下条件的三位数：该数是11的倍数，并且个、十、百位数字各不相同。

程序编写如下：

```
#include  <iostream.h>
void main( )
{
     int  i, x, y, z;
     for(i=100; i<1000; i++)
      if(i%11==0)                       // 11的倍数
      {
         x=i/100;                       // 取得百位的数值
         y=i%100/10;                    // 取得十位的数值
         z=i%10;                        // 取得个位的数值
         if(x!=y && y!=z && z!=x)
            cout << i << endl;
     }
}
```

我们在程序中利用了整数的整除和模运算将一个整数分解，分别取得它的个位、十位和百位。这种方法效率较高，而且不易出错。

**例4.20**　使用牛顿迭代法，求方程$x^3+2x^2+3x+4=0$在1附近的实数根。

牛顿迭代法是利用曲线的导数方程就是曲线的切线方程这一数学原理，通过不断迭代求得切线方程与$x$坐标轴的交点，来逼近精确解，最终求得曲线方程的近似解。其推导后的迭代公式如下：

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$$

程序编写如下：

```
#include  <iostream.h>
#include  <math.h>
void main( )
{
   float  x, x1, f, f1;
   x=x1=1;                              // 初始化迭代因子
   do
   {
```

```
        x=x1;
        f=x*x*x+2*x*x+3*x+4;          //原方程
        f1=3*x*x+2*2*x+3;             //导数方程
        x1=x-f/f1;                    //迭代公式
    }while(fabs(x1-x)>=1e-9);
    cout << "方程的根为: " << x1 << endl;
}
```

在程序中当前后两次计算结果之间的差值小于$10^{-9}$时，我们就认为已经接近精确解了，继续计算对结果的影响不大，这时我们就退出循环，终止程序的运行。

## 习题4

1. 从键盘输入3个数，输出其中最大的数。
2. 画流程图求解1+2+3+…+1000。
3. 编程求解1000以内，可以同时被9和11整除的所有数据。要求按照一行5个数据，按行输出。
4. 编写程序，将从键盘输入的百分制成绩转换为五分制成绩并显示在屏幕上。成绩转换规则如下：A对应90~100分，B对应80~89分，C对应70~79分，D对应60~69分，E对应60分以下。请使用if-else语句和switch语句两种方法求解。
5. 每个公民在生活中都需要依法纳税，而个人所得税的税率是根据收入的多少按不同比例缴纳的，税率设置如下：收入少于800元的部分不计税，收入大于800元的部分，按照超出部分分段计税。超出部分的计算具体规定如下：

| | | |
|---|---|---|
| | 超出部分 <= 500 | 超出部分按5%缴税 |
| 500 < | 超出部分 <= 2000 | 超出部分中超过500元至2000元的部分按10%缴税，其他如上 |
| 2000 < | 超出部分 <= 5000 | 超出部分中超过2000元至5000元的部分按15%缴税，其他如上 |
| 5000 < | 超出部分 <= 20000 | 超出部分中超过5000元至20000元的部分按20%缴税，其他如上 |
| 20000 < | 超出部分 <= 40000 | 超出部分中超过20000元至40000元的部分按25%缴税，其他如上 |
| 40000 < | 超出部分 <= 60000 | 超出部分中超过40000元至60000元的部分按30%缴税，其他如上 |
| 60000 < | 超出部分 <= 80000 | 超出部分中超过60000元至80000元的部分按35%缴税，其他如上 |
| 80000 < | 超出部分 <= 100000 | 超出部分中超过80000元至100000元的部分按40%缴税，其他如上 |
| 100000 < | 超出部分 | 超出部分中超过100000元的部分按45%缴税，其他如上 |

请编程实现对任意输入的收入金额都可以立刻计算出应缴个人所得税数额，并且将应缴税额显示在屏幕上。请用if语句编程实现上述功能。

6. 求符合以下条件的所有的四位数：该数各位数字的四次方之和等于该数本身。
7. 编程序求解1000以内的所有素数。
8. 用循环语句输出如下图形：

```
*******
 *****
  ***
   *
  ***
 *****
*******
```

9. 使用穷举法（三重循环）求解例4.19。
10. 编程求解下列分数序列的前30项之和：

$$\frac{2}{1}, \frac{3}{2}, \frac{5}{3}, \frac{8}{5}, \frac{13}{8}, \frac{21}{13}, \cdots$$

# 第5章 函 数

## 5.1 概述

一个较大的程序为便于实现一般应分为若干个程序模块，每一个模块实现一个特定的功能。在C++语言中，由函数实现模块的功能。函数是C++程序的构成基础。一个C++程序可由一个主函数main( )和若干个函数构成。函数的实现，将有利于数据共享，节省开发时间，增强程序的可靠性。本章将介绍函数的定义、说明以及调用等重要内容。此外还将介绍有关作用域的概念，使读者能够了解变量、函数的作用域及生存期，进而提高对变量和函数使用的灵活性。

先举一个简单的函数调用的例子。

**例5.1** 简单函数调用。

```
#include <iostream.h>    //Li0501.cpp
void printstar( )
{  cout<<"******************"<<'\n';    }
void print_message( )
{  cout<<"  Welcome to C++"<<'\n';      }
void main( )
{  printstar( );
   print_message( );
   printstar( );
}
```

运行结果为:

```
******************
  Welcome to C++
******************
```

printstar和printf_message是用户定义的函数名，分别用来输出一行“*”号和一行信息。

任何一个C++程序都是从主函数，即从main( )的开花括号开始执行，一直到main( )的闭花括号为止。在此过程中，如果遇到一个函数调用语句，则暂时中断main( )函数的执行，将流程转到被调函数，执行完被调函数再返回到主函数中断处，继续执行。

## 5.2 函数的定义与调用

### 5.2.1 函数的定义

函数可以是系统预定义的，也可以是用户自定义的。前者称为系统函数或标准库函数，后者称为用户自定义函数。

一个函数必须定义后才能使用。所谓定义函数，就是编写完成函数功能的程序块。一个C++函数由函数头与函数体两部分组成，其一般形式如下:

```
[<类型>] <函数名> ([形式参数列表])          // 函数头
{
        语句                                 // 函数体
}
```

方括号内的<类型>说明可以省略，下同。

为了说明函数的结构，请看下面的例子。

**例5.2** 求两个数中的最小值。

```
#include <iostream.h>                    //Li0502.cpp
int min(int x, int y)                    // 函数定义: 求两个数中的最小值
{   int z;
    z=x<y?x : y;
    return ( z );
}
void main( )
{  int a, b, c;
   cout<<"请输入两个整数:"<<endl;
   cin>>a>>b;
   c=min(a, b);                          // 函数调用
   cout<<"最小值为: "<<c<<endl;
}
```

运行情况如下:

```
请输入两个整数:
6 8
最小值为: 6
```

### 1. 函数头

函数头的组成形式如下:

```
[<类型>] <函数名> ([形式参数列表])
```

函数类型规定函数返回值的类型。如int min(int x, int y)则表示函数min将返回一个int类型的值。若<类型>缺省，则表示函数返回值为int型。无返回值的类型是void类型。

函数名是函数的标识，它应是一个有效的C++标识符。

形式参数简称形参。形参列表是包含在圆括号中的0个或多个以逗号分隔的变量定义。它规定了函数将从调用函数中接收几个数据及它们的类型。之所以称为形参，是因为在定义函数时系统并不为这些参数分配存储空间，只有被调用时，向它传递了实际参数（简称实参）才为形参分配存储空间。如上例，变量a、b是实参，而变量x、y是形参。

### 2. 函数体

一个函数体是用花括号括起来的语句序列。它描述了函数实现一个功能的过程，并要在最后执行一个函数返回，返回的作用是:

(1) 将流程从当前函数返回其上级（调用函数）。

(2) 撤销函数调用时为各参数及变量分配的内存空间。

(3) 向调用函数返回最多一个值。

一般来说，函数返回由返回语句来实现。如例5.2中的return(z); 就可以执行上述三个功能。

return语句的一般形式为:

```
return 表达式;  或  return (表达式); 或 return;
```

具有表达式的返回语句实现过程如下:

(1) 先计算出表达式的值。

(2) 如果表达式的类型与函数的类型不相同时，将表达式的类型自动转换为函数的类型，这种转换是强制性的。

(3) 将计算出的表达式的值返回到调用处作为调用函数的值。

（4）将程序执行的控制由被调函数转向调用函数，执行调用函数后面的语句。

关于return语句的使用说明:

（1）有返回值的return语句，用它可以返回一个表达式的值，从而实现函数之间的信息传递。

（2）无返回值的函数须用void来说明函数类型。该函数中可以有return语句，也可以无return语句。当被调函数中无return语句时，程序执行完函数体的最后一条语句后返回调用函数，相当于函数体的右括号有返回功能。

（3）一个函数体中可以有多个return语句，但每次只能通过一个return语句执行返回操作。

**例5.3**　返回一个整数的绝对值。

```
int abs(int x)              //Li0503.cpp
{   if(x>=0)
        return x;
    else
        return -x;
}
```

一个函数体中有多个return语句时，如上述函数中有两个return语句，每个return语句的返回值类型都应与函数定义一致。

函数可以只执行一个功能而不向调用函数返回任何值，如一个输出，例5.1中的printstar( )函数就是这样。这时return语句后的表达式是空的，它只执行将流程返回以及撤销函数中的参数变量的存储空间的操作。空的return语句位于函数末尾时可以缺省，由函数体后的花括号执行返回功能。

### 5.2.2　函数的调用

调用函数是实现函数功能的手段。C++中的函数调用很丰富。它不仅有传值调用，而且还有引用调用。另外，C++中还允许设置形参的缺省值等。

#### 1. 函数的传值调用

函数的调用是用一个表达式来表示的。其调用形式如下:

```
<函数名> (实参列表);
```

其中，实参列表是由0个、1个或多个实参构成，多个参数之间用逗号分隔，每个参数是一个表达式。实参是用来在调用函数时给形参初始化的，一般地，要求在函数调用时，实参的个数和类型要与形参的个数和类型一致，即个数相等，类型相同。实参对形参初始化是按其位置对应进行，即第一个实参的值赋给第一个形参，第二个实参的值赋给第二个形参，依此类推。如例5.2中的值传递如图5-1所示。相当于在函数参数传递时有int x=a; int y=b; 。

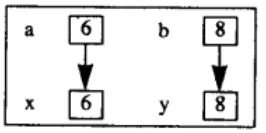


图5-1　参数传递

使用传值调用方式时，实参可以是常量、变量和表达式。系统先计算实参表达式的值，再将实参的值按位置对应地赋给形参，即对形参进行初始化。因此传值调用的实现是系统将实参拷贝一个副本给形参，形参和实参分别占用不同的存储单元。在被调函数中，形参可以被改变，但这只影响副本中形参的值，而不影响调用函数中实参的值。所以传值调用的特点是形参值的改变不影响实参。

下面举一个传值调用的例子。

**例5.4**　理解传值调用。

```
#include <iostream.h>  //Li0504.cpp
void swap(int x, int y)
{   int t;
    t=x;    x=y; y=t;
    cout<<"x="<<x<<","<<"y="<<y<<endl;
}
```

```
void main( )
{   int a(4), b(5);    // 等价于int a=4, b=5;
    swap(a, b);
    cout<<"a="<<a<<","<<"b="<<b<<endl;
}
```

运行结果为:

```
x=5,y=4
a=4,b=5
```

从该程序的结果看，在swap( )函数中，形参x和y的值交换了，而在main( )中，实参a和b的值并没有发生变化，可见swap( )函数中形参值的改变，并没有影响实参a和b的值。

如果想使形参的改变影响实参就不能用这种传值调用，而应选用引用调用。

### 2. 函数的引用调用

引用是一种特殊类型的变量，可以认为是另一个变量的别名。通过引用名与通过被引用的变量名访问变量的效果是相同。

引用运算符&用来说明一个引用。例如: int i, &r=i;

这里运算符&用来说明为对变量的引用，其引用类型为int。经这样说明后，变量i与引用r代表的是同一个变量。因此，对引用的操作就是对原变量的操作。例如:

```
int i=10, &r=i;
r+=10;
```

其结果是变量i的值增加为20。

C++引入引用的主要目的是为方便函数间数据的传递，在应用中主要是作为函数的参数。

将上例改写为引用调用:

```
#include <iostream.h>  // Li0504_1.cpp 理解引用调用
void swap(int &x, int &y)
{   int t;
    t=x; x=y; y=t;
    cout<<"x="<<x<<","<<"y="<<y<<endl;
}
void main( )
{   int a(4), b(5);
    cout<<"交换前:  "<<endl;
    cout<<"a="<<a<<","<<"b="<<b<<endl;
    swap(a, b);
    cout<<"交换后:  "<<endl;
    cout<<"a="<<a<<","<<"b="<<b<<endl;
}
```

运行结果为:

```
交换前:
a=4,b=5
x=5,y=4
交换后:
a=5,b=4
```

程序运行结果表明实参a、b的内容交换成功。这是因为函数的参数是引用，所以在函数中直接对引用指向的变量进行操作。

### 5.2.3 函数的参数

#### 1. 函数参数的求值顺序

当一个函数带有多个参数时，标准C++语言没有规定在函数调用时实参的求值顺序。而编译器根据对代码进行优化的需要自行规定对实参的求值顺序。有的编译器规定自左至右，有的编译器规定自右至左，这种对求值顺序的不同规定，对一般参数来讲没有影响。但是，如果实参表达式中存在带有副作用的运算符时，就有可能产生由于求值顺序不同而造成的二义性。例如，下面是一个由于使用不同求值顺序的编译器造成二义性的例子。

**例5.5** 使用不同求值顺序的编译器造成二义性的示例。

```
#include <iostream.h>  // Li0505.cpp
int fun(int a, int b)
{  return b; }
void main( )
{  int x(5),y(6);
   int z=fun(x--, x+y);
   cout<<z<<endl;
}
```

在该程序中，调用表达式:

```
z=fun(x--, x+y);
```

其中，实参是两个表达式x--和x+y。如果编译器对实参的求值顺序是自左至右的，x+y的值为10，结果z的值也为10。如果编译器对实参的求值顺序是自右至左的，x+y的值为11，z的值也为11。z的值由于实参值的不同，调用fun( )函数后，返回值也不同，于是便造成了这个程序在不同编译器下输出不同结果的二义性。克服这种二义性的方法是改变fun( )函数的两个实参的写法，尽量避免二义性的出现。如main( )函数可改写如下:

```
void main( )
{ int x(5), y(6);
  int w=x--;
  int z=fun(w, x+y);
  cout<<z<<endl;
}
```

对函数fun( )的两个实参表达式，无论怎样的求值顺序结果都是一样的。这样就避免了二义性的出现。在VC 6.0中的求值顺序是自右往左的。

#### 2. 设置函数参数的默认值

在C++语言中，允许在函数的定义或说明时给一个或多个参数指定默认值。但是，要求指定了默认值的参数后，其右边不能出现没有指定默认值的参数，即默认值集中在参数的右边。例如:

```
int fun1(int a, int b=1, int c=2);  合法
int fun1(int a=1, int b, int c=2);  不合法
```

在函数调用时，编译器按自左至右的顺序将实参与形参结合，当实参的数目不足时，编译器将按同样的顺序用定义中或说明中的默认值来补足所缺少的实参。例如，对定义int fun1(int a, int b=1, int c=2);，下列三种调用方式是等价的:

```
fun1(3); fun1(3, 1); fun1(3, 1, 2);
```

**例5.6** 设置函数参数的默认值。

```
#include <iostream.h> // Li0506.cpp
void fun(int x=1, int y=2, int z=3)
```

```
{  cout<<"x="<<x<<","<<"y="<<y<<","<<"z="<<z<<endl;    }
void main( )
{   fun( );
    fun(5);
    fun(5, 6);
    fun(5, 6, 7);
}
```

该程序执行结果如下:

```
x=1, y=2, z=3
x=5, y=2, z=3
x=5, y=6, z=3
x=5, y=6, z=7
```

该程序中在函数定义时设置了参数的默认值，而在调用该函数时，有的无实参，有的实参数目不足，有的实参数目与形参相等，在此分若干不同情况说明了默认值的使用。

## 5.3 函数的原型说明

在C++中，当函数定义在前，调用在后时，调用前可以不必说明；如果一个函数的定义在后，调用在前，此时调用前必须说明。

按照上述原则，凡是被调函数都在调用函数之前定义，则可以对函数不加说明。但是，这样做在安排程序中函数的顺序上要花费很多精力，在复杂的调用中，一定要考虑好谁先谁后，否则将发生错误。为了避免安排函数定义的顺序，并使程序逻辑结构清晰，常常将主函数放在程序开头，这样就需要在主函数之前对被调函数进行原型说明。

函数原型说明的一般格式如下:

```
[<类型>] <函数名> ([形式参数列表]);
```

这种说明类似于定义函数时的函数头。这里，类型是该函数返回值的类型，括号中的参数说明可以仅给出每个参数的类型，也可以指明每个形参名及其类型。说明函数原型的目的是告诉编译程序，该函数的返回值类型、参数个数和各个参数的类型，以便其后调用该函数时，编译程序对该函数调用时的参数的类型、个数、顺序及函数的返回值进行有效性检查。下面是一个使用原型说明的例子。

**例5.7** 使用原型说明。

```
#include <iostream.h>                    //Li0507.cpp
void main( )
{   int min(int, int);                   //函数原型说明，也可以放在main( )函数之前
    int a, b, c;
    cin>>a>>b;
    c=min(a, b);                         //函数调用
    cout<<"最小值为: "<<c<<endl;
}
int min(int x, int y)                    //函数定义: 求两个数中的最小值
{   int z;
    z=x<y?x:y;
    return(z);
}
```

上例中的原型说明也可以写成:

```
int min(int x, int y);
```

注意，在C++中函数的原型说明是一个说明语句，其后的分号不可缺少；函数的原型说明可出现在

程序中的任何位置，且对一函数的原型说明次数没有限制。函数原型中可以只依次说明参数类型而不给出形参名的原因在5.8节变量的作用域中说明。如果函数参数指定默认值，则函数原型说明时要带默认值，如对例5.6中函数的原型说明为“void fun(int x=1, int y=2, int z=3);”。

**例5.8**　验证哥德巴赫猜想：一个大偶数可以分解为两个素数之和。试编程序，将 96 到 100 之间的全部偶数分解成两个素数之和。

```
#include <iostream.h>  //Li0508.cpp
#include <iomanip.h>
#include <math.h>
int prime(int a)
{   int z, i, k;
    z=0;
    k=(int)sqrt(a);
    for (i=2; i<=k; i++)
          if (a%i==0) break;
    if (i>=k+1)  z=1;
    return(z);
}
void main( )
{  int a, b, m, n=0;
   for (m=96; m<=100; m=m+2)
   {      for (a=2; a<=m/2; a++)
          {      if (prime(a))
                 {      b=m-a;
                        if (prime(b))
                        {      cout<<setw(3)<<m<<"="<<setw(2)<<a<<"+"<<setw(2)<<b<<"  ";
                               n++;
                               if(n%5==0) cout<<endl;  //控制一行输出5组数据
                        }
                 }
          }
   }
}
```

运行结果为:

```
96= 7+89      96=13+83      96=17+79      96=23+73      96=29+67
96=37+59      96=43+53      98=19+79      98=31+67      98=37+61
100= 3+97     100=11+89     100=17+83     100=29+71     100=41+59
100=47+53
```

## 5.4　函数的嵌套调用和递归调用

### 5.4.1　函数的嵌套调用

C++语言的函数定义都是互相平行的、独立的。一个函数定义内部包含另一个函数定义，称为嵌套定义。C++语言规定不能嵌套定义函数，但可以嵌套调用函数，所谓嵌套调用指的是在调用一个函数的过程中，又调用另一个函数，见图5-2。

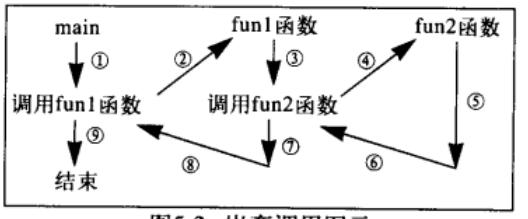


图5-2　嵌套调用图示

图5-2表示的两层嵌套（包含main函数共3层），其执行过程如下:

① 执行main函数的开头部分。

② 遇调用函数fun1的语句，流程转去执行fun1函数。

③ 执行fun1函数的开头部分。

④ 遇调用函数fun2的语句，流程转去执行fun2函数。

⑤ 执行fun2的函数体。

⑥ 返回调用fun2函数处，即返回fun1函数。

⑦ 继续执行fun1函数尚未执行的部分，直到fun1函数结束。

⑧ 返回main函数中调用fun1函数处。

⑨ 继续执行main函数的剩余部分直到结束。

嵌套调用是经常使用的。下面举例说明函数嵌套调用的使用。

**例5.9** 编写程序求两个数a、b的最大公约数和最小公倍数。辗转相除法（又称欧几里得算法）如下：（1）求a对b的余数；（2）如余数r为0，则b是最大公约数，算法结束；否则执行下一步；（3）将除数作为新的被除数，余数作为新的除数，即执行a=b,b=r，转（1）。

```
#include <iostream.h>                    //Li0509.cpp
int gys(int x, int y)
{    int r;
     while ((r=x%y)!=0)
     {      x=y; y=r; }
     return(y);
}
int gbs(int x, int y)
{    int bs, ys;
     ys=gys(x, y);                       //调用求最大公约数的函数
     bs=x*y/ys;
     return bs;
}

void main( )
{   int x, y, g, bs;
    cout<<"请输入两个整数: ";
    cin>>x>>y;
    g=gys(x, y);
    bs=gbs(x, y);
    cout<<"最大公约数为:"<<g<<'\n'<<"最小公倍数为:"<<bs<<'\n';
}
```

运行情况如下：

```
请输入两个整数: 21 28<Enter>
最大公约数为:7
最小公倍数为:84
```

**例5.10** 编写程序，输入k和n，求出多项式 $\sum_{i=1}^{n} i^k$ 的值。

```
#include <iostream.h>  //Li0510.cpp
int sum_of_power(int k, int n), power(int m, int n);
void main( )
{   int k, n;
    cout<<"请输入k和n:";
    cin>>k>>n;
    cout<<"sum of "<<k<<" power of integers from 1 to "<<n<<"=";
```

```
    cout<<sum_of_power(k, n)<<endl;
}
int sum_of_power(int k, int n)
{   int i, sum(0);
    for (i=1; i<=n; i++)
          sum+=power(i, k);
    return sum;
}
int power(int m, int n)
{   int i, product(1);
    for(i=1; i<=n; i++)
         product*=m;
    return product;
}
```

运行情况如下:

```
请输入k和n: 5 10<Enter>
sum of 5 power of integers from 1 to 10=220825
```

**例5.11**　用弦截法求方程$x^4+4x^3-3x^2+5x+6=0$的根。方法如下:

(1) 确定求值区间。输入两个不同点$x_1$、$x_2$，直到f($x_1$)和f($x_2$)异号为止。注意$x_1$、$x_2$的值不应相差太大，以保证($x_1$, $x_2$)区间内只有一个根。

(2) 连接f($x_1$)和f($x_2$)两点，此线(称为弦)交$X$轴于$x$，见图5-3。

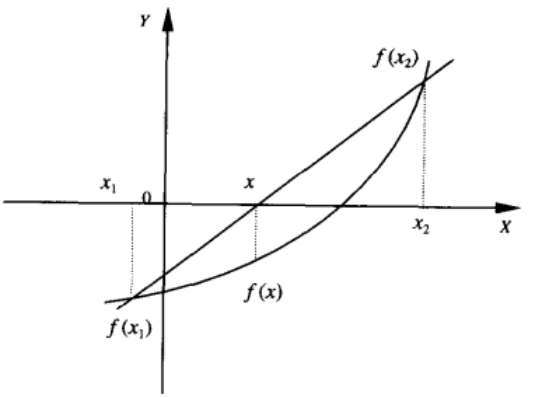



图5-3　弦截法示意图

$x$点坐标可用下式求出:

$$x=\frac{x_1\cdot f(x_2)-x_2\cdot f(x_1)}{f(x_2)-f(x_1)}$$

再求出f($x$)。

(3) 若f($x$)与f($x_1$)同号，则根必在($x$, $x_2$)区间内，此时将$x$作为新的$x_1$。如果f($x$)与f($x_2$)同号，则表示根在($x_1$, $x$)区间内，将$x$作为新的$x_2$。

(4) 重复步骤(2)和(3)，直到|f($x$)|<ε为止，ε为一个很小的数，例如$10^{-6}$。此时认为f($x$)≈0。

分别用几个函数来实现各部分的功能:

(1) 用f($x$)来求$x$的函数: $x^4+4x^3-3x^2+5x+6=0$。

(2) 用函数xpoint($x_1$, $x_2$)来求f($x_1$)和f($x_2$)的连线与$X$轴的交点$x$的坐标。

(3) 用函数root($x_1$, $x_2$)来求($x_1$, $x_2$)区间的实根。显然执行root函数的过程中要用到函数xpoint，而执行xpoint函数的过程中要用到f函数。

```
#include <iostream.h>                    //Li0511.cpp
#include <math.h>
double f(double x);
double xpoint(double x1, double x2);
double root(double x1, double x2);
void main( )                             //主函数
{   double x1, x2, x, f1, f2;
    do                                   //输入两个不同点x1、x2，直到f(x1)和f(x2)异号为止
    {    cout<<"input x1, x2:"<<endl;
```

```
        cin>>x1>>x2;
        f1=f(x1);
        f2=f(x2);
    }while( (f1*f2)>=0 );
    x=root(x1, x2);
    cout<<"A root of equation is "<<x<<endl;
}
double f(double x)                          // 定义函数，求f(x)=x⁴+4x³-3x²+5x+6
{  double y;
   y=(((x+4.0)*x-3.0)*x+5.0)*x+6.0;
   return(y);
}
double xpoint(double x1, double x2)   // 定义xpoint函数，求出弦与X轴交点
{  double y;
   y=(x1*f(x2)-x2*f(x1))/(f(x2)-f(x1));
   return(y);
}
double root(double x1, double x2)     // 定义root函数，求近似根
{  double x, y, y1;
   y1=f(x1);
   do
   {    x=xpoint(x1, x2);
        y=f(x);
        if ( y*y1>0 )
        {       y1=y;
                x1=x;
        }
        else
                x2=x;
   }while( fabs(y)>=1e-6);
   return(x);
}
```

运行情况如下:

```
input x1, x2:
10 -2<Enter>
A root of equation is -0.692532
```

从上例中可以看到:

(1) 在定义函数时，函数名为f、xpoint、root的3个函数是相互独立的，并不相互从属。这3个函数均定义为双精度型。

(2) 在main函数外对这3个函数进行了原型说明。

(3) 程序从main函数开始执行。先执行一个do-while循环，作用是：输入x1和x2，判别f(x1)和f(x2)是否异号，如果不是异号则重新输入x1和x2，直到满足f(x1)与f(x2)异号为止。然后用函数调用root(x1, x2)求根x。调用root函数的过程中，要调用xpoint函数来求f(x1)与f(x2)连线的交点x。在调用xpoint函数的过程中要用到函数f来求x1和x2的相应的函数值f(x1)和f(x2)。这就是函数的嵌套调用，见图5-4。

(4) 在root函数中要用到求绝对值的函数fabs，它是对实型数求绝对值的标准数学库函数。因此在文件开头用#include <math.h>，即把数学库函数的函数原型说明等信息包含进来。

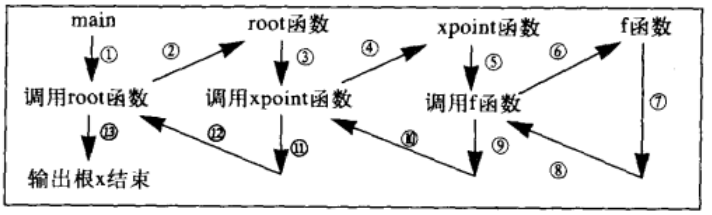



图5-4　求方程的根嵌套调用图示

### 5.4.2　函数的递归调用

在一个函数的执行过程中直接或间接地调用该函数本身，称为函数的递归调用。C++语言中允许函数的递归调用。

在调用函数f的过程中，又要调用f函数，这是直接调用本函数，称为直接递归调用，见图5-5。

在调用函数f1的过程中要调用函数f2，而在函数f2的执行过程中又要调用函数f1，这是间接调用函数f1，称为间接递归调用，见图5-6。

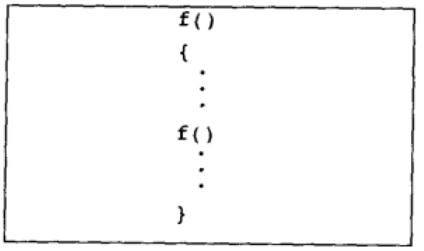



图5-5　函数的直接递归调用

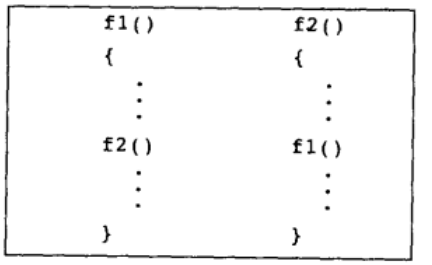



图5-6　函数的间接递归调用

程序中不应该出现无终止的递归调用，而只应出现有限次的、有终止的递归调用，这可以用if语句来控制，只有在某一条件成立时才继续执行递归调用，否则就不再继续。下面举例说明递归调用的过程。

**例5.12**　递归计算$n!$的函数。

$n!$可以递归地描述为：

$$fact(n)=\begin{cases}\text{非法} & (n<0)\\ 1 & (n=0\text{或}n=1)\\ n*fact(n-1) & (n>1)\end{cases}$$

```
#include <iostream.h> // Li0512.cpp
#include <stdlib.h>
float fac(int);
void main( )
{   int n;
    float y;
    cout<<"input an integer number:"<<endl;
    cin>>n;
    y=fac(n);
    if(n>=0)
          cout<<n<<"!="<<y<<endl;
}
float fac(int n)
{   float f;
    if (n<0)
    {     cout<<"n<0, data error!"<<endl;       return (-1);
```

```
    }
    else if (n==0|| n==1) f=1;
    else f=fac(n-1)*n;
    return(f);
}
```

运行情况如下:

```
input an integer number:
4<Enter>
4!=24
```

递归调用的具体过程如图5-7所示。

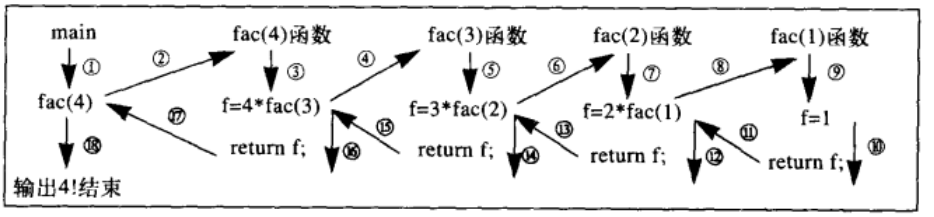



图5-7 求$n!$的递归调用图示

**例5.13** 用递归实现求两个数的最大公约数。用前面介绍的辗转相除法，求两个数的最大公约数的过程可以递归地描述为:

$$gys(a,b)=\begin{cases} b & (a\%b==0) \\ gys(b,\ a\%b) & (a\%b!=0) \end{cases}$$

```
#include <iostream.h>      // Li0513.cpp
int gysfun(int, int);      // 在所有函数外进行函数原型说明
void main( )
{   int a, b, gys;
    cout<<"输入两个整数a, b"<<endl;
    cin>>a>>b;
    gys=gysfun(a, b);
    cout<<a<<"和"<<b<<"的最大公约数为: "<<gys<<endl;
}
int gysfun(int a, int b)
{   if(a%b==0) return b;
    else return gysfun(b, a%b);
}
```

运行情况如下:

```
输入两个整数a, b
24  32<Enter>
24和32的最大公约数为: 8
```

**例5.14** 汉诺塔（Tower of Hanoi）问题。问题是: 古代有一个梵塔，塔内有3根钻石做的柱子，其中1根柱子上有64个金子做的盘子。64个盘子从下到上按照由大到小的顺序叠放。僧侣的工作是把这64个盘子从第1根柱子上移动到第3根柱子上，移动的规则如下:

(1) 每次只能移动一个盘子。

(2) 移动的盘子必须放在其中一根柱子上。

（3）大盘子在移动过程中不能放在小盘子上。

我们的目标是编写一个程序，该程序可以打印出将盘子从第1根柱子转移到第3根柱子每一步移动的顺序。

用递归来思考:

（1）考虑第1根柱子上只有1个盘子的情况，这样盘子可以从第1根柱子直接移动到第3根柱子上。

（2）考虑第1根柱子上有2个盘子的情况。

① 把第1个盘子从柱子1移动到柱子2上。

② 把第2个盘子从柱子1移动到柱子3上。

③ 把第1个盘子从柱子2移动到柱子3上。

（3）考虑第1根柱子包含3个盘子的情况，这样一直推广到64个盘子的情况（实际上，可以推广到任意数目的盘子）。

假设柱子1上有3个盘子:

① 为了把盘子3移动到柱子3上，前两个盘子必须先移动到柱子2上。

② 将盘子3从柱子1移动到柱子3上。

③ 为了把前两个盘子从柱子2移动到柱子3上，使用相同的策略。这一次要把柱子1作为中间柱子。

把这个问题推广到64个盘子的情形:

① 将上面的63个盘子从柱子1移动到柱子2上。

② 再把盘子64从柱子1移动到柱子3上。

③ 现在，前面的63个盘子都在柱子2上。为了把盘子63从柱子2移动到柱子3上，首先要把前62个盘子从柱子2移动到柱子1上，接着再把盘子63从柱子2移动到柱子3上。按照相似的过程移动剩下的62个盘子。

经过上面的讨论，得到该递归算法如下:

假设第1根柱子上有$n$个盘子，并且$n>1$。

（1）以柱子3作为中间柱子，把前$n-1$个盘子从柱子1移动到柱子2上。

（2）把盘子$n$从柱子1移动到柱子3上。

（3）以柱子1作为中间柱子，把前$n-1$个盘子从柱子2移动到柱子3上。

```
#include <iostream.h>  // Li0514.cpp
void hanoi(int, char, char, char);
void move(char, char);
void main( )
{  int m;
   cout<<"input the number of diskes:"<<endl;
   cin>>m;
   cout<<"The step to moving " <<m<<" diskes:\n";
   hanoi(m, 'A', 'B', 'C');
}
void hanoi( int n, char one, char two, char three)
{  if (n==1)
   move(one, three);
   else
   {    hanoi(n-1, one, three, two);
        move(one, three);
        hanoi(n-1, two, one, three);
   }
}
void move(char getone, char putone)
{  cout<<getone<<"-->"<<putone<<endl;  }
```

运行情况如下:

```
input the number of diskes:
3<Enter>
The step to moving 3 diskes:
A-->C
A-->B
C-->B
A-->C
B-->A
B-->C
A-->C
```

请读者自己分析例5.13和例5.14中递归调用的执行过程。

何时采用递归解决问题呢? 要满足如下两个条件:

(1) 能化小问题规模。可把要解决的问题转化为一个新的问题,而新的问题的解法仍与原来的解法相同,只是所处理的对象有规律地递增或递减,例: $n!=n*(n-1)!$。可应用这个转化过程使问题得到解决。

(2) 能确定终结条件。必须有一个明确的结束递归的条件,例: $n==0$||$n==1$时,阶乘的值为1。

## 5.5 内联函数

函数调用会降低程序的执行效率,因为调用函数实际上是将程序执行顺序转到被调函数所存放的内存地址,函数执行完毕后,再返回调用函数继续执行。这种转移操作在转去前要保护现场并记忆执行的地址,转回后先要恢复现场,并按原来保存的地址继续执行。因此,函数调用要有一定的时间和空间开销,于是影响其效率。特别是对一些函数体代码不是很大,但又频繁地被调用的函数来说,解决其效率问题尤为重要,C++中引入内联函数就是为了解决这一问题的。

编译程序处理内联函数的方法是,将程序中出现的内联函数的调用表达式替换为内联函数的函数体语句。显然,将函数体的代码替换到程序中,会增加目标程序代码量,进而增加程序空间的开销,但它可以避免函数调用的时间开销,因此它是以空间代价来换取时间代价的。

内联函数的定义方法很简单,只要在函数定义前加上关键字inline即可。例如:

```
inline int min(int x, int y)
{ return(x<y?x:y);}
```

其中inline是关键字。函数min是一个内联函数。

**例5.15** 编程求1~10中各个数的立方。

```
#include <iostream.h> //Li0515.cpp 理解内联函数
inline int cube(int x)
{   return(x*x*x); }
void main( )
{   int c;
    c=cube(4);
    cout<<4<<"*"<<4<<"*"<<4<<"="<<c<<endl;
    c=cube(8);
    cout<<8<<"*"<<8<<"*"<<8<<"="<<c<<endl;
    c=cube(12);
    cout<<12<<"*"<<12<<"*"<<12<<"="<<c<<endl;
}
```

运行结果为:

```
4*4*4=64
```

```
8*8*8=512
12*12*12=1728
```

该程序中，函数cube( )是一个内联函数，其特点是该函数在编译时被替换，而不是像一般函数那样是在运行时被调用。

内联函数具有一般函数的特性，它与一般函数所不同之处仅在于函数调用的处理。一般函数进行调用时，要将程序执行转到被调函数中，然后再返回到调用它的函数中；而内联函数在调用时，是将调用表达式用内联函数体来替换。在使用内联函数时，应注意以下几点：

（1）在内联函数内不允许出现循环语句、switch语句及复杂嵌套的if语句。

（2）内联函数的定义必须出现在内联函数第一次被调用之前。形参和实参的关系与一般的函数相同。

（3）对于用户指定的内联函数，编译器是否作为内联函数来处理由编译器自行决定。说明内联函数时，只是请求编译器当出现这种函数时，作为内联函数的扩展来实现，而不是命令编译器这样做。

（4）内联函数的实质是用空间来换取时间的方法，当出现多次调用同一内联函数时，程序本身占用的空间将有所增加。但是，在上例中，内联函数仅调用一次时，并不增加程序所占的存储空间。

## 5.6 函数重载

所谓函数重载是指同一个函数名可以对应着多个函数的实现。例如，可以给函数名add( )定义多个函数实现，该函数的功能是求和，即求两个操作数的和。其中，一个函数实现是求两个整数之和，另一个实现是求两个浮点数之和，再一个实现是求两个复数之和。每种实现对应着一个函数体，这些函数的名字相同，但是函数的参数类型不同，这就是函数重载的概念。

函数重载要求编译器能够唯一地确定调用一个函数时应执行哪个函数代码，即采用哪个函数实现。确定函数实现时，要求从函数参数的个数和类型来区分。这就是说，进行函数重载时，要求同名函数的参数个数不同，或参数类型不同。否则，将无法实现重载。

### 5.6.1 参数类型不同的重载函数

下面举一个参数类型不同的重载函数的例子。

**例5.16** 求两个操作数之和。

```
#include <iostream.h>  //Li0516.cpp
int add(int, int);
double add(double, double);
void main( )
{  cout<<add(2, 3)<<endl;
   cout<<add(2.0, 3.5)<<endl;
}
int add(int x, int y)
{  return(x+y);  }
double add(double x, double y)
{  return(x+y);  }
```

在该程序中，main( )函数中调用相同名字add的两个函数，前面一个add( )函数对应的是两个int型数据求和的函数实现，而后面一个add( )函数对应的是两个double型数据求和的函数实现。这便是函数的重载。

该程序的运行结果为：

```
5
5.5
```

### 5.6.2 参数个数不同的重载函数

下面举一个参数个数不同的重载函数的例子。

**例5.17** 找出几个int型数中的最大者。

```
#include <iostream.h>// Li0517.cpp 参数个数不同的重载函数
int max(int a, int b);
int max(int a, int b, int c);
void main( )
{   cout<<max(5, 8)<<endl;
    cout<<max(10, 4, 20)<<endl;
}
int max(int a,int b)
{   return (a>b?a:b);}
int max(int a, int b, int c)
{   int m=a;
    if(b>m) m=b;
    if(c>m) m=c;
    return m;
}
```

该程序中出现了函数重载，函数名max对应两个不同的实现，函数的区分是依据参数个数的不同，这里的两个函数实现中，参数个数分别为2和3，在使用函数时根据实参的个数来选取不同的函数实现。

## 5.7 使用C++系统函数

C++的编译系统提供了很多函数供编程者调用。本节将介绍使用系统函数的方法。

C++系统将所提供的系统函数的说明分类放在不同的.h文件（又称为头文件）中。例如，有关数学常用函数，如求绝对值函数、求平方根函数和三角函数等，放在math.h文件中；判断字母、数字、大写字母、小写字母等函数放在ctype.h文件中；有关字符串处理函数放在string.h文件中；等等。因此，编程者在使用系统函数时应注意：

（1）了解C++系统提供了哪些系统库函数。不同的C++编译系统所提供的系统函数不同；同一种C++编译系统的不同版本所提供的系统函数的多少也不一定相同。只有了解系统所提供的系统函数后，才能根据需要选用。

阅读C++编译系统的使用手册可以了解该系统所提供的系统函数，手册中会给出各种系统函数的功能、函数的参数和返回值以及函数的使用方法。另外，也可以通过联机帮助了解一些系统函数的简单情况。

（2）必须知道某个系统函数的说明在哪个头文件中。因为要调用某个系统函数，必须将该系统函数的说明所在的头文件包含在调用的程序中，否则连接不上，将出现连接错。例如，当使用sqrt( )函数求某个数的平方根时，就需要在程序中，将math.h头文件包含进去。

（3）调用一个函数时，一定要将该函数的功能、参数的返回值弄清楚，否则难以正确调用该函数。

**例5.18** 将输入的三个字符转换成大写字符后输出。

```
#include <iostream.h>      // Li0518.cpp
#include <ctype.h>         // A
#define N 3
void main( )
{   int i;
    char c;
    for(i=0;i<N;i++)
    {     cin>>c;
          if(islower(c))   // 判断字符c是否是小写字符，是小写字符返回1，否则返回0
                c-=32;
           cout<<c<<'\t';
```

```
    }
    cout<<endl;
}
```

如果缺少A行，编译时则出现错误：“error C2065: 'islower' : undeclared identifier”。因为这里使用了islower函数，所以要包含ctype.h头文件。

## 5.8 作用域和存储类别

作用域即作用范围，它是指所定义的标识符在哪一个区间内有效，即在哪一个区间内可以使用。在程序中出现的各种标识符，它们的作用域是不同的。C++把作用域分为五类：块作用域、文件作用域、函数原型作用域、函数作用域和类作用域。类作用域在以后相关章节介绍，本节只介绍前四种作用域。

存储类别决定了何时为变量分配存储空间及该存储空间所具有的特征。在定义变量时，指定变量的存储类别。

### 5.8.1 作用域

标识符只能在说明它或定义它的范围内进行存取，而在该范围之外是不可以进行存取的。下面介绍不同的标识符的作用域是如何规定的。

#### 1. 块作用域

C++中把用花括号括起来的一部分程序称为块。在一个块中说明的标识符，其作用域从说明点开始到该块结束为止。例如：

```
float f1(int a)                          // 函数f1
{
          int b, c;          }  a,b,c有效
          .                  |
          .                  |
          .                  }
          .                  |
}
char f2(int x, int y)                    // 函数f2
{
     int b, c;
     int i, j;          }     x,y,b,c,i,j有效
     .                  |
     .                  |
     .                  }
}
void main( )                             // 主函数
{
     int m, n;          }     m, n有效
     .                  |
     .                  |
     .                  }
}
```

在一个函数内部定义的变量或在一个块中定义的变量称为局部变量。如上例中的所有变量均为局部变量。换句话说就是块作用域的变量均为局部变量。在一个函数内定义的局部变量，在退出函数时，局部变量就不存在了；在块内定义的变量，在退出该块时，块作用域内的局部变量也就不存在了。

**例5.19** 将3个数从大到小排序。

```
#include <iostream.h>          // Li0519.cpp
```

```
void swap(int &a, int &b)
{   if(a<b)
    {     int t;              //t是局部变量，退出该复合语句，则t不能使用
          t=a; a=b; b=t;
    }
}
void main( )
{   int a, b, c;
    cout<<"请输入三个整数："<<endl;
    cin>>a>>b>>c;
    swap(a,b);
    swap(a,c);
    swap(b,c);
    cout<<a<<'\t'<<b<<'\t'<<c<<endl;
}
```

说明:

(1) 形参也是局部变量。例如f1函数中的形参a，只是在f1函数内有效，其他函数不能调用。

(2) 主函数main中定义的变量m、n只在主函数中有效，f1函数中定义的b、c只在f1函数中有效，f2函数中定义的b、c、i、j只在f2函数中有效。

(3) 不同函数中可以使用相同名字的变量，它们代表不同的对象，互不干扰。例如，在f1函数中定义了变量b、c，在f2函数中也定义了变量b、c，它们在内存中占不同的存储单元，互不混淆。

(4) 具有块作用域的标识符在其作用域内，将屏蔽包含本块的同名标识符，即局部定义优先。下面的例子说明了局部变量的同名问题。

**例5.20** 理解局部定义优先。

```
#include <iostream.h>          //Li0520.cpp
void main( )
{   int a=1, b=2;
    ++a;
    ++b;
    {     int b=4, c;
          c=a+b;               // c只能在该复合语句内使用
          cout<<"a="<<a<<", "<<"b="<<b<<", "<<"c="<<c<<endl;
    }
    cout<<"a="<<a<<", "<<"b="<<b<<endl;
}
```

块B

块A

根据以上规则，块B中定义的变量b屏蔽了块A内的变量b。因此在块B内使用变量b是使用的块B内定义的变量b，而不是使用块A内定义的变量b。一旦退出块B，块B内定义的变量b、c就不存在了。因此程序运行结果如下:

```
a=2, b=4, c=6
a=2, b=3
```

(5) 在for语句中说明的循环控制变量具有块作用域，其作用域是包含for语句的那个内层块，而不是仅仅作用于for语句。例如:

```
{
   ...
          for(int i=0; i<10; i++)
          {     cout<<i<<'\t';
          }
```

```
        cout<<'i='<<i<<endl;
        }
```

等价于

```
    {
    ...
    int i;
        for(i=0; i<10; i++)
        {     cout<<i<<'\t';
        }
        cout<<'i='<<i<<endl;
    }
```

这种变量的说明不同于在循环体内说明的变量。又如:

```
#include <iostream.h>
void main(void)
{    for(int i=0; i<10; i++)     // A
     {     int j=0;
           cout<<++j<<'\t';
     }                           // B
}
```

其输出结果为:

```
1    1    1    1    1    1    1    1    1    1
```

因为变量j的作用域为从A行开始到B行结束，每一次循环开始时，为变量j分配存储空间，而执行到B行时，结束变量j的作用域，变量j不复存在。所以尽管在循环体内每一次都对j加1，但输出的值均为1。

### 2. 文件作用域

文件是C++的编译单位。文件作用域是在所有函数外说明的，其作用域从说明点开始，一直延伸到源文件结束。

通常把超出一个函数的作用域称为全局作用域，其他几种不超出一个函数的作用域称为局部作用域。

在函数外定义的变量或用extern说明的变量称为全局变量。全局变量的缺省作用域是：从定义全局变量的位置开始到该源程序文件结束，即符合标识符先定义后使用的原则。当全局变量出现先引用后定义时，要用extern对全局变量进行外部说明，其方法在后面介绍。当在块作用域内的变量与全局变量同名时，局部变量优先。在块作用域内可通过作用域运算符“::”来引用与局部变量同名的全局变量。

**例5.21**　在块作用域内引用文件作用域的同名变量。

```
#include <iostream.h>              //Li0521.cpp
int i=10;
void main( )
{  int i, j=5;
   i=20;                           //访问局部变量i
   ::i=::i+4;                      //访问全局变量i
   j=::i+i;                        //访问全局变量i和局部变量i、j
   cout<<"::i="<<::i<<endl;
   cout<<"i="<<i<<endl;
   cout<<"j="<<j<<endl;
}
```

程序运行结果为:

```
::i=14
```

```
i=20
j=34
```

### 3. 函数原型作用域

在函数原型的参数列表中说明的参数名，作用域只在该函数原型内，称为函数原型作用域。因此，在函数原型中说明的标识符可以与函数定义中说明的标识符不同。由于所说明的标识符与该函数的定义及调用无关，因此可以在函数原型说明中只进行参数的类型说明，而省略参数名。例如:

```
float max(float a, float b);      // 函数max的原型说明
...
float max(float x, float y)       // 函数max的定义
{
      ...
}
```

由于可以省略函数原型说明中的参数名，因此函数max( )的原型说明也可以写成:

```
float max(float, float);
```

### 4. 函数作用域

作为goto语句转移目标的标志——标号具有函数作用域。在本函数内所给出的标号，无论它在什么地方，都可以用语句goto引用它。但是不能用goto语句把流程转到其他函数体内。语句标号是唯一具有函数作用域的标识符。例如:

```
void f1( )
{
{ label1:  ...}
...
    if(...) goto label1;
    if(...) goto label2;          //A
}
void f2( )
{
    label2: ...
}
void main( )
{
    ...
}
```

编译时，指出A行的标号没有定义。

## 5.8.2 存储类别

前面已经介绍了，从变量的作用域角度来分，可以分为全局变量和局部变量。而从变量值存在的时间角度来分，可以分为静态存储变量和动态存储变量。存储类别规定了变量的存在时间（即生存期），变量值的存在时间，就是变量空间存在的时间。

一个C++源程序经编译和连接后，产生可执行程序。要执行该程序，系统必须为程序分配内存空间，并将程序装入所分配的内存空间内。一个程序在内存中占用的存储空间可以分为三个部分：程序区、静态存储区和动态存储区，如图5-8所示。

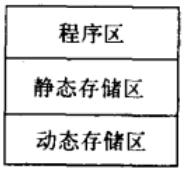


图5-8 程序在内存占用的存储空间

程序区用来存放可执行程序的程序代码，而为变量分配静态存储区还是动态存储区是由变量的存储类别所决定的，变量的存储类

别是由程序设计者根据程序设计的需要指定的。

在C++中，变量的存储类别有以下四种:

- auto（自动）
- static（静态）
- extern（外部）
- register（寄存器）

在说明或定义时，存储类别说明符应放在数据类型说明符之前。格式如下:

```
[<存储类别>] <类型> <标识符>[=<初始化表达式>];
```

例如:

```
auto int i=0;
```

定义了变量i为自动变量，类型为整型，初值为0。

### 1. 自动变量（auto）

自动变量被分配在动态存储区中。函数的形参和块作用域中定义的变量都属此类，在函数执行期间，当执行到变量作用域开始处时，系统动态地为变量分配存储空间，而当执行到作用域结束处，系统收回这种变量所占用的存储空间。因此这类局部变量称为自动变量。自动变量用关键字auto作存储类别说明。例如:

```
int f(int a)
{   auto int b, c=2;          // 定义b、c为自动变量
    .
    .
    .
}
```

关键字“auto”可以省略，即变量的缺省存储类别是auto。前面介绍的函数中定义的变量都没有存储类别说明，因此均缺省指定为自动变量。在本例的函数体中: `auto int b, c=2;`与`int b, c=2;`等价。

注意，对于自动变量，若没有明确地赋初值，其初值是不确定的。如上例中的b没有确定的初值。

### 2. 静态变量（static）

静态变量被分配在静态存储区中。有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值，即占用的存储单元不释放，在下一次调用该函数时，该变量已有值，就是上一次函数调用结束时的值。这时就应该指定该局部变量为“静态存储类别”，用关键字static进行说明。通过下面简单的例子可以了解它的特点。

**例5.22**　考察静态局部变量的值。

```
#include <iostream.h> // Li0522.cpp
fun (int x)
{   static int a=3;
    a+=x;
    return(a);
}
void main( )
{   int k=2, m=1, n;
    n=fun(k);
    cout<<"first: n="<<n<<endl;
    n=fun(m) ;
    cout<<"second: n="<<n<<endl;
}
```

程序运行结果为:

```
first: n=5
second: n=6
```

在第一次调用函数fun时a的初值为3，第一次调用结束时，a的值为5。由于a是静态局部变量，在函数调用结束后，它并不释放，仍保留a=5；在第二次调用fun函数时，a的初值为5（上次调用结束时的值）。

对静态局部变量的说明:

（1）静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在程序整个运行期间始终存在。

（2）静态局部变量的初始化仅在程序开始执行时处理一次，在程序运行时它的值始终存在。以后每次调用函数时不再重新赋初值，而只是保留上次函数调用结束时的值。而自动变量赋初值是在函数调用时进行的，每调用一次函数重新赋一次初值，相当于执行一次赋值语句。

（3）定义静态局部变量缺省初值为0（对数值型变量）或空字符（对字符变量）。

（4）虽然静态局部变量在函数调用结束后仍然存在，但其他函数是不能引用它的。

若要保留函数上一次调用结束时的值，则需要用静态局部变量。例如可以用如下方法求$n!$。

**例5.23** 打印1到5的阶乘值。

```
#include <iostream.h>  // Li0523.cpp
int fac(int n);
void main( )
{  int i;
   for (i=1; i<=5; i++)
          cout<<i<<"!="<<fac(i)<<endl;
}
int fac(int n)
{  static int f=1;
   f=f*n;
   return(f);
}
```

程序运行结果为:

```
1!=1
2!=2
3!=6
4!=24
5!=120
```

每次调用fac(i)，打印出一个i!，同时保留这个i!的值以便下次再乘（i+1）。

如果将上例中的static去掉，程序运行结果变为:

```
1!=1
2!=2
3!=3
4!=4
5!=5
```

显然结果是不正确的。原因是将变量f说明成自动变量后，每调用一次fac函数，变量f都重新赋值为1。

### 3. 外部变量（extern）

外部变量（即全局变量）是在函数的外部定义的，它的作用域为从变量的定义处开始，到本程序文件的末尾。在此作用域内，全局变量可以为程序中各个函数所引用。编译时将外部变量分配在静态存储区。全局变量的缺省存储类别是extern。

有时需要用extern来说明外部变量，以扩展外部变量的作用域:

**(1) 在一个文件内说明外部变量。**

如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件尾部。如果在定义点之前的函数想引用该变量，则应该在引用之前用关键字extern对该变量进行“外部变量说明”，表示该变量是一个已经定义的外部变量。有了此说明，就可以从“说明”处起，合法地使用该外部变量，如下所示。

**例5.24** 用extern说明外部变量，扩展外部变量的作用域。

```
#include <iostream.h>                    //Li0524.cpp
int min(int a, int b)                    //定义min函数
{   int c ;
    c=a<b?a:b ;
    return(c);
}
void main( )
{   extern a, b;                         //A   说明外部变量
    cout<<min(a, b)<<endl;
}
int a=3, b=5;                            //B   定义外部变量
```

在本程序文件的B行定义了外部变量a、b。由于外部变量定义的位置在函数main之后，因此本来在main函数中不能引用外部变量a、b。现在由于在main函数的A行用extern对a、b进行了“外部变量说明”，表示a、b是已经定义的外部变量，这样在main函数中就可以合法地使用全局变量a、b。如果不用extern说明，编译时会出错，系统认为a、b未定义。一般做法是将外部变量的定义放在引用它的所有函数之前，这样可以避免在函数中再加一个extern说明。

用extern说明外部变量时，类型名可以不写。例如：`extern  a;`

**(2) 在多文件的程序中说明外部变量。**

一个C++程序可以由一个或多个源程序文件组成。如果程序由多个源程序文件组成，那么在一个文件中想引用另一个文件中已定义的外部变量，可以用下面介绍的方法解决。

如果一个程序中包含两个文件，在两个文件中都要用到同一个外部变量num，不能分别在两个文件中各自定义外部变量num，否则在进行程序的连接时会出现“重复定义”的错误。正确的做法是：在任一个文件中定义变量num，而在另一个文件中用extern对num进行“外部变量说明”，即：

```
extern num;
```

在编译和连接时，系统会由此知道num是一个已在其他地方定义的外部变量，并将在另一文件中定义的外部变量的作用域扩展到本文件，在本文件中可以合法地引用它。

例如，有两个程序f1.cpp和f2.cpp，C行定义变量、分配存储空间，A行说明变量、不分配存储空间，这两行中的变量是同一个变量；而B行及D行定义各自的变量a、b，它们各自分配不同的存储空间。

f1.cpp

```
extern int x, y; // A
main()
{......}
static int a, b; // B
f1()
{......}
```

f2.cpp

```
int x, y; // C
f2()
{......}
static int a, b; // D
f3()
{......}
```

下面举例来说明这种引用。

**例5.25** 用extern将外部变量的作用域扩展到其他文件。

本程序的作用是给定b的值，输入a和m，求a × b和$a^m$的值。本程序由两个程序文件Li0525_1.cpp和Li0525_2.cpp构成。

Li0525_1.cpp中的内容为：

```
#include <iostream.h>
int a;                                   //定义外部变量
void main( )
```

```
{   int power(int);                    // 对调用函数作说明
    int b=3, c, d, m;
    cout<<"enter the number a and its power m:"<<endl;
    cin>>a>>m;
    c=a*b;
    cout<<a<<"*"<<b<<"="<<c<<endl;
    d=power(m);
    cout<<a<<" ** "<<m<<"="<<d<<endl;
}
```

Li0525_2.cpp文件中的内容为:

```
extern a;                              // 说明a是一个已定义的外部变量
power (int n)
{   int i, y=1;
          for(i=1; i<=n; i++)
                y=y*a;
    return (y);
}
```

可以看到Li0525_2.cpp文件中的开头有一个extern说明，说明在本文件中出现的变量a是一个已经在其他文件中定义过的外部变量，本文件不必再次为它分配内存，可以直接使用该变量。假如程序有4个源程序文件，在一个文件中定义外部整型变量a，其他3个文件都可以引用a，但必须在其他3个文件中都加上一个“extern a;”说明。在各文件经过编译后，再将各目标文件连接成一个可执行的目标文件。

但是使用这样的外部变量应十分谨慎，因为在执行一个文件中的函数时，可能会改变外部变量的值，它会影响到另一文件中函数的执行结果。

extern既可以用来扩展外部变量的作用域，又可以使外部变量的作用域从一个文件扩展到程序中的其他文件，那么系统是如何区别处理的呢？实际上，在编译时遇到extern时，先在本文件中寻找外部变量的定义，如果找到，就在本文件中扩展作用域。如果找不到，就在连接时从其他文件寻找外部变量的定义，如果找到，就将作用域扩展到本文件。如果找不到，按出错处理。

**(3) 用static说明外部变量。**

有时在程序设计中希望某些外部变量只限于被本文件引用，而不能被其他文件引用。这时可以在定义外部变量时加一个static说明。例如:

f1.cpp

```
static int a;
void main()
{
      .
      .
      .
}
```

f2.cpp

```
extern a;
fun(int n)
{
      .
      .
      .
       a=a*n;
      .
      .
      .
}
```

在f1.cpp中定义了一个全局变量a，但它用static说明，因此只能用于本文件，虽然在f2.cpp文件中用了“extern a;”，但f2.cpp文件中无法使用f1.cpp中的全局变量a。这种加上static说明、只能用于本文件的外部变量称为静态外部变量。

需要指出的是对外部变量加static说明，并不意味着这时才是静态存储（存放在静态存储区中），而

不加static的是动态存储（存放在动态存储区中）。两种形式的外部变量都是静态存储方式，只是作用范围不同而已，都是在编译时分配内存的。

#### 4. 寄存器变量（register）

一般情况下，变量的值是存放在内存中的。当程序中用到哪一个变量的值时，由控制器发出指令将内存中该变量的值送到运算器中，经过运算器进行运算，如果需要保存数据，再从运算器将数据送到内存存放。

如果有一些变量使用频繁，则为了节省时间，可以将这些变量存放在CPU的寄存器中，需要用时直接从寄存器中取出参加运算，不必再到内存中去取。由于对寄存器的存取速度远远快于对内存的存取速度，因此这样做可以提高执行效率。这种变量叫做“寄存器变量”，用关键字register说明。

由于register与auto说明的变量仅存储位置上不同，并且现代的C++编译器能决定哪些变量存放在寄存器中，因此register说明符已很少使用。

## 5.9 程序的多文件组织

我们在编制大型程序时，为了方便程序的设计与调试，往往将一个程序分成若干模块，把实现相关功能的程序和数据放在一个文件中。当一个完整的程序中的若干函数被存放在两个及两个以上文件中时，称为程序的多文件组织。这种多文件组织的程序如何进行编译和连接？一个文件中的函数如何调用另一个文件中的函数？这是本节要介绍的内容。

### 5.9.1 内部函数和外部函数

函数本质上是全局的，因为一个函数要被另外的函数调用，但是，也可以指定函数不能被其他文件调用。根据函数能否被其他源文件调用，将函数分为内部函数和外部函数。

#### 1. 内部函数

如果一个函数只能被本文件中的其他函数所调用，则它称为内部函数。在定义内部函数时，在函数名和函数类型的前面加上static即可。形式如下：

```
static <类型标识符> <函数名> (<形参列表>)
```

如：`static int fun(int x, int y)`

使用内部函数，可以使函数只局限于所在文件。如果在不同的文件中有同名的内部函数，则互不干扰。这样，不同的人可以编写不同的函数，而不必担心所用函数是否会与其他文件中的函数同名。通常把只能由同一文件使用的函数和外部变量放在一个文件中，在它们前面都加上static使之局部化，其他文件不能引用。

#### 2. 外部函数

在多文件组织的程序中，一个源程序文件中定义的函数，不仅能在本源程序文件中使用，而且可以在其他源程序文件中使用，这种函数称为外部函数。在定义外部函数时，在函数名和函数类型的前面加上extern即可。形式如下：

```
extern int fun(int x, int y)
```

这样，函数fun就可以被其他文件调用。C++语言规定，如果定义函数时省略extern，则隐含规定为外部函数。本书前面所用的函数均为外部函数。如：

```
extern int max(int x, int y)
{...}
int min(int x, int y)
{...}
```

函数max( )和min( )均为外部函数。

在多文件组织方式中，在一个文件中要调用在另一个程序文件中定义的外部函数前，必须对被调用的函数进行原型说明，并在函数原型说明的前面加上关键字extern。例如，在程序文件z1.cpp中定义了函

数fun( )，在程序文件z2.cpp中要调用文件z1.cpp中已定义了的函数fun( )，则在调用前增加函数原型说明。

z1.cpp

```
int fun(int x, int y)
{......}
```

z2.cpp

```
extern int fun(int,int);
......
k=fun(x,y);
```

**例5.26** 用如下公式计算排列函数。

$$p(n,k)=\frac{n!}{(n-k)!}$$

在Li0526_1.cpp文件中定义求阶乘的函数，在Li0526_2.cpp文件中调用求阶乘函数。

Li0526_1.cpp程序如下:

```
int fac(int n)          //返回n!=n*(n-1)*(n-2)*...*2*1
{    if(n<0) return 0;
     int f=1;
     while(n>1)
          f*=n--;
     return f;
}
```

Li0526_2.cpp程序如下:

```
#include <iostream.h>
extern int fac(int);
                    //在Li0526_1.cpp中定义了函数，在Li0526_2.cpp中调用要加原型说明
void main( )
{   int n, k, p;
    cout<<"请输入n和k:";
    cin>>n>>k;
    p=fac(n)/fac(n-k);
    cout<<"p("<<n<<","<<k<<")="<<p<<endl;
}
```

运行情况如下:

```
请输入n和k: 5  2<Enter>
p(5,2)=20
```

### 5.9.2 多文件组织的编译和连接

当一个完整的程序由多个源程序文件组成时，如何将这些文件进行编译并连接成一个可执行的程序文件呢？不同的计算机系统，处理方法可能是不同的。通常有以下几种处理方法:

(1) 用包含文件的方式，在定义main( )函数的文件中将组成同一程序的其他文件用包含指令包含进来（关于文件包含的意义请参看第6章），由编译程序对这些源程序文件一起编译，并连接成一个可执行文件。这种方法适用于编写较小的程序，不适用于编写大的程序，因为对任何一个文件中的微小修改，均要重新编译所有的文件，然后才能连接。

(2) 使用工程文件的方法，将组成一个程序的所有文件都加到工程文件中，由编译器自动完成多文件组织的编译和连接。如在Visual C++中，可以有多种方法建立工程文件。一种方法是先为包含main( )函数的文件建立一个工程文件，然后选择主菜单上的“Project”菜单中的子菜单“Add to project”的下一级子菜单“File”，将其他程序文件加入到工程文件中。这时对多个文件的编译和连接方法，与一个文件组成一个程序的方法完全相同。有关工程文件的更详细的说明，可查阅相关资料。

设计大型程序时，建议使用这种方法。当修改某一个源程序文件后，编译器只需对已修改的源程序文件重新编译，而没有必要对其他源程序文件重新编译，因此可以大大提高编译和连接的效率。

## 习题5

1. 编写被调函数求四个数的平均值。
2. 编写被调函数统计输入的整数序列中的奇数个数和偶数个数。
3. 下列公式用来计算笛卡儿平面上的两点：$(x_1, y_1)$和$(x_2, y_2)$间的距离为：

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

给定圆心和圆上一点的坐标，利用上述公式计算圆的半径。编写程序输入圆心和圆上一点的坐标，然后输出圆的半径、直径、周长和面积，要求用函数实现求两点间的距离。

4. 组合函数C(n, k)在给定的n个元素的集合中求不同的k个元素的子集的个数。该函数可以用以下公式计算：

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

5. 编写一个函数，用下面的公式求$e^x$的近似值。要求最后一项小于$10^{-6}$。

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

6. 编写一个函数，求出1000以内的素数，要求每行输出5个素数。
7. 一个数如果恰好等于它的因子之和，这个数就称为“完数”。例如，6的因子为1、2、3，而6=1+2+3，因此6是“完数”，编写程序找出1000之内的所有完数，并按下面的格式输出其因子：

```
6  its factors are:  1  2  3
```

8. 编写一个程序通过重载求两个数中小数的函数，分别求两个实数和两个整数中的小数。
9. 编写一个内联函数实现求出3个数中的最大数。
10. 定义一个带默认参数值的函数求$n!$，$n$的缺省值为10。在主函数中调用两次该函数，一次给出实参，另一次不给实参，体会函数默认值的意义。
11. 编写一个递归函数。将这个整数的每个位上的数字按相反的顺序输出。例如：输入1234，输出4321。
12. 用递归函数实现求Fibonacci数列的前$n$项，$n$作为函数的参数。Fibonacci数列的公式为：

$$fib(n) = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ fib(n-1) + fib(n-2) & n > 2 \end{cases}$$

13. 编写一个程序，用递归方法求$n$阶勒让德多项式的值，递归公式为

$$p_n(x) = \begin{cases} 1 & (n = 0) \\ x & (n = 1) \\ ((2n-1)xp_{n-1}(x) - (n-1)p_{n-2}(x))/n & (n > 1) \end{cases}$$

14. 编写一个程序，输入一个十进制数，输出相应的二进制数。设计一个递归函数实现数制转换。
15. 编写两个函数分别输出以下两个图形。

```
*            *
**          ***
***        *****
****      *******
*****    *********
```

16. 用矩形法求积分：

$$\int_a^b (1 + x)x\mathrm{d}x$$

# 第6章　编译预处理

C++语言允许在程序中用预处理命令写一些命令行。这些预处理命令不是C++语言本身的组成部分，不能直接对它们进行编译，必须在对程序进行通常的编译之前，先对这些特殊的命令进行“预处理”。预处理行都以“#”开头，末尾不加分号以区别于C++语句。它们可以出现在程序中的任何位置，但一般写在程序的首部。C++提供的预处理功能主要有以下三种：宏定义、文件包含和条件编译。分别用宏定义命令、文件包含命令、条件编译命令来实现。

## 6.1　宏定义

### 6.1.1　不带参数的宏定义

不带参数的宏定义就是用一个宏名来代表一个字符串，它的一般形式为：

```
#define 宏名 宏体
```

宏体是一个字符串，宏名是合法的C++标识符，如：

```
#define PI 3.1415926
```

经过定义的宏名可应用在程序中。它的作用是指定用宏名PI来代表“3.1415926”这个字符串，在编译预处理时，将程序中在该命令以后出现的所有的PI都代换成“3.1415926”。在预编译时将宏名代换成字符串的过程称为“宏代换”。

**例6.1**　求球体的表面积及体积。

```
#include <iostream.h> // Li0601.cpp
#define PI 3.1415926
void main( )
{   double r, s, v;
    cout<<"input radius :";
    cin>>r;
    s=PI*r*r;
    v=4.0/3*PI*r*r*r;
    cout<<"\n面积为: "<<s<<"\n体积为: "<<v<<'\n';
}
```

说明：

（1）宏名一般习惯用大写表示，以便与变量名相区别。但这并非是规定的，也可用小写字母。

（2）使用宏名代表一个字符串，可以减少程序中的重复书写。例如，如果不定义PI代表3.1415926，则在程序中要多处书写3.1415926，不仅麻烦，而且在输入程序时也容易产生错误，用宏名代替，简单且不易出错，因为记住一个宏名（它的名字往往用容易理解的单词表示）比记住一个字符串容易，而且读程序时能立即知道它的含义。

（3）当需要改变某一个变量的值时，可以只改变#define命令行，一处改处处改。例如，如果已经将PI定义为3.1415926，现要将PI的值全部改为3.14，可以用

```
#define PI 3.14     //程序中出现PI的地方全部改为3.14
```

（4）宏定义用宏名代替一个字符串，只进行简单的代换，不进行任何计算，也不进行正确性检查。如果加了分号，则会连分号一起进行代换。如：

```
#define PI 3.1415926;
area=PI * r * r;
```

经过宏代换后，该语句为

```
area=3.1415926; * r * r;
```

显然出现语法错误。

(5) #define命令的有效范围为定义之后到源文件结束。

```
#include <iostream.h>
void main( )
{   void f1( );
    #define PI 3.1415926
    f1( );
}
void f1( )
{   cout<<PI<<endl;
}
```

通常，#define命令写在文件开头，函数之前，作为文件的一部分，在此文件内有效。

(6) 可以用#undef命令终止宏定义的作用域。例如:

```
#define  PI  3.1415926
void main( )
{
      .
      .
      .
}
#undef  PI
f1( )
{
      .
      .
      .
}
```




由于#undef的作用，使PI的作用范围在#undef行处终止，因此在f1函数中，PI不再代表3.1415926。这样可以灵活控制宏定义的作用范围。

(7) 在进行宏定义时，可以引用已定义的宏名，进行层层代换。

**例6.2** 求圆的周长和面积。

```
#include <iostream.h>  // Li0602.cpp
#define R 5.0
#define PI 3.1415926
#define L 2*PI*R
#define S PI*R*R
void main( )
{
   cout<<"L="<<L<<"\nS="<<S<<endl;
}
```

运行结果为:

```
L=31.4159
S=78.5398
```

经过宏代换后L被代换为2*3.1415926*5.0，S代换为3.1415926*5.0*5.0。

（8）对程序中用双引号引起来的字符串内的字符，即使与宏名相同，也不进行代换。如例6.2中双引号内的L和S不代换，而双引号外的L和S被代换。

### 6.1.2 带参数的宏定义

用宏定义还可以定义带参数的宏，它的一般形式为：

```
#define 宏名(参数表) 宏体
```

其中，宏名是一个标识符，参数表中可以有一个参数，也可以有多个参数，多个参数之间用逗号分隔，宏体是被代换用的字符序列。在代换时，首先进行参数代换，然后再将代换后的字符串进行宏代换。例如：

```
#define ADD(x, y)  x+y
```

如在程序中出现如下语句：

```
sum=ADD(2, 4);
```

则被代换成：

```
sum=2+4;
```

如果程序中出现如下语句：

```
sum=ADD(a+2, b+3);
```

则被代换为：

```
sum=a+2+b+3;
```

可见，宏代换只是将字符串中出现的、与宏定义时参数表中参数名相同的字符序列代换。

**例6.3** 带参数的宏定义。

```
#include <iostream.h>          //Li0603.cpp
#define MUL(a, b) a*b
void main( )
{    int x(5), y(8), t;        //x初值为5，y初值为8
     t=MUL(x+1, y-2);
     cout<<"t="<<t<<endl;
}
```

进行宏代换后，主函数的内容如下：

```
void main( )
{   int x(5), y(8), t;   //x初值为5，y初值为8
    t= x+1*y-2;
    cout<<"t="<<t<<endl;
}
```

执行该程序后，输出结果为：

```
t=11
```

使用带参数的宏定义时要注意如下几点：

（1）带参数的宏定义的字符串应写在一行上，如果需要写在多行上时，在每行结束时，使用续行符（“\”）结束，并在该符号后按下回车键，最后一行除外。

（2）在书写带参数的宏定义时，宏名与左括号之间不能出现空格，否则空格右边的字符都作为替代字符串的一部分。例如：

```
#define ADD  (x, y)  x+y
```

这时宏名ADD被认为是不带参数的宏定义，它代表字符串“(x,y) x+y”。

（3）定义带参数的宏时，字符串中与参数名相同的字符序列适当地加上圆括号是十分重要的，这样可以避免代换后在优先级上产生问题。例如:

```
#define SQR(x) x*x
```

当程序中出现下列语句:

```
m=SQR(a+b);
```

代换后应为:

```
m=a+b*a+b;
```

如果想代换后变为:

```
m=(a+b)*(a+b);
```

则应该这样进行宏定义:

```
#define SQR(x) (x)*(x)
```

可见，在宏体中加圆括号将会改变代换后的表达式的优先级。

又如，对上述的宏定义，程序中的语句:

```
m=100/SQR(5);
```

代换后为:

```
m=100/(5)*(5);
```

结果m的值为100。

而对下述宏定义:

```
#define SQR(x) ((x)*(x))
```

上述语句代换后

```
m=100/((5)*(5));
```

结果m的值为4。

进一步看出，在所定义的字符串中适当地加圆括号所起的作用。

读者可以看出带参数的宏定义与函数之间有一定类似之处，在调用函数时也是在函数名后的括号内写实参，也要求实参与形参的数目相等。但带参的宏定义与函数是不同的，主要区别有:

（1）函数调用是在程序运行时处理的，分配临时的内存单元。而宏代换则是在编译时进行的，在代换时并不分配内存单元，不进行值的传递，也没有“返回值”的概念。

（2）函数调用时，先求出实参表达式的值，然后赋值给形参。而使用带参的宏只是进行简单的字符代换。例如上面的SQR(a+b)，在宏代换时并不求a+b的值，而只将实参“a+b”代替形参x。

（3）对函数中的形参和实参都要定义类型，且二者的类型要求一致，如不一致，系统自动进行类型转换。而宏不存在类型问题，宏名没有类型，它的参数也没有类型，只是一个符号代表，代换时代入指定的字符即可。

（4）宏代换不占运行时间，只占编译时间。而函数调用则占运行时间（分配单元、保留现场、值传递、返回）。

在C++中，带参数的宏定义常用内联函数实现，因为内联函数的定义和调用形式与一般函数一致，但实际执行时没有函数调用，速度较快。

## 6.2 “文件包含”处理

文件包含的作用是让编译预处理器把另一个源文件嵌入（包含）到当前源文件中的该指令处。文件包含命令格式如下:

```
#include <文件名>
```

或

```
#include "文件名"
```

图6-1表示“文件包含”的含义。图6-1a为文件file1.cpp，它有一个#include "file2.h"命令，然后还有其他内容（用A表示）。图6-1b为另一文件file2.h，文件内容用B表示。在编译预处理时，要对#include 命令进行“文件包含”处理：将file2.h 的全部内容复制插入到#include "file2.h"命令处，即file2.h被包含到file1.cpp中，得到图6-1c所示的结果。在编译中，将“包含”以后的file1'.cpp（即6-1c所示）作为一个源文件单位进行编译。

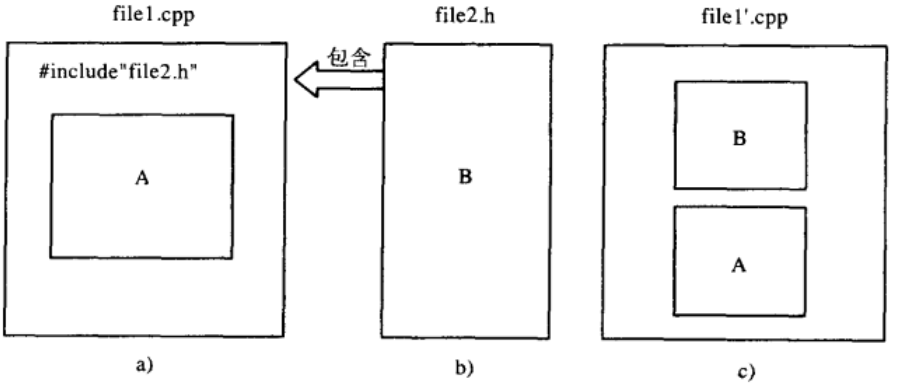



图6-1 文件包含的含义

“ #include <文件名> ”格式一般用于嵌入C++系统提供的头文件，这些头文件一般存于C++系统标准头文件包含目录中，如VC 6.0的标准头文件包含目录为C:\Program File\Microsoft Visual Studio\VC98\include。在程序中若有#include <math.h>，则C++预处理器遇到这条命令后，就自动到该include目录中搜寻math.h，并把它嵌入到当前文件中。这种搜寻方式称为标准方式。“ #include "文件名" ”格式则首先在文件所在的当前目录中进行搜寻；如果搜寻不到，再按标准方式进行搜寻。如文件在D盘当前目录中，用前一种方式时，系统就直接到标准头文件包含目录中去搜寻，所以，这种方式适用于包含系统提供的头文件。而用后一种方式时，则要先在D盘当前目录中搜寻，搜寻不到再到系统标准头文件包含目录中去搜寻，从效率上看，它适用于包含用户建立的头文件。

所谓“头文件”是指一些存放着标准库函数的原型说明，或者存放着符号常量、类型定义、类和其他复杂类型的定义以及与程序环境相关信息的.h文件，即文件扩展名为h。头文件有系统提供的，例如，前面程序中已出现过的iostream.h便是一个系统提供的有关输入/输出操作信息的头文件。头文件也有根据用户需要编写的。文件包含命令一般放在程序头部。因为被包含文件的内容是被插入到该文件包含命令出现的位置，因此，被包含文件的内容插在文件头，以便后面程序对它们的引用。

在使用文件包含时应注意如下几点:

(1) 一条文件包含命令只能包含一个文件，若想包含多个文件须用多条文件包含命令。例如:

```
#include <iostream.h>
#include <math.h>
    .
    .
    .
```

（2）被包含文件中还可以使用文件包含命令。文件包含命令可以嵌套使用。例如，定义一个头文件，其名字为myfile1.h，该文件的内容如下：

```
#include "myfile2.h"
    .
    .
    .
```

myfile2.h文件的内容如下：

```
#include "myfile3.h"
    .
    .
    .
```

上述嵌套包含见图6-2，它的作用与图6-3相同。

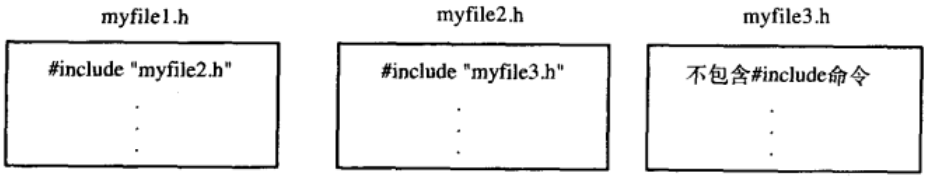



图6-2　文件嵌套包含示意图

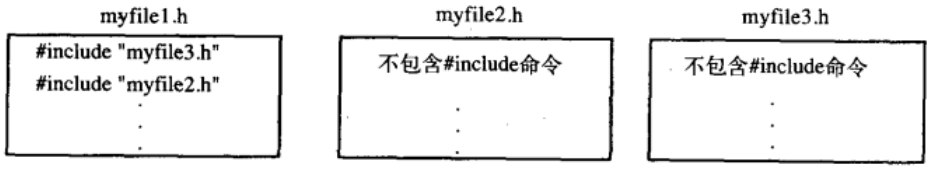



图6-3　文件嵌套包含示意图

## *6.3　条件编译

利用条件编译可以使同一个源程序在不同的条件下产生不同的目标代码，用于完成不同的功能。利用条件编译可在调试程序时增加一些调试语句，以达到跟踪的目的，但程序调试好后，重新编译时，使调试语句不参与编译。常用的条件编译命令有如下三种格式。

### 1. 格式1

```
#ifdef  宏名
        程序段1
[#else
        程序段2]
#endif
```

上述格式中方括号里的内容可缺省。程序段可以由若干条预处理命令和语句组成。其功能为：如果宏名已被定义，则编译程序段1，否则编译程序段2。这种条件编译对于提高C++源程序的通用性是很有好处的。一个C++源程序在不同的计算机系统上运行时，不同的计算机系统间有一定的差异。例如，有的计算机系统以16位（2个字节）来存放一个整数，而有的则以32位（4个字节）来存放一个整数。针对一种系统编写的程序，若要移植到另一个系统中运行，往往需要对源程序进行必要的修改，这就降低了程序的通用性。可以用以下的条件编译来编写通用程序。

```
#ifdef  COMPUTER
```

```
        #define INTEGER_SIZE 16
#else
        #define INTEGER_SIZE 32
#endif
```

即如果COMPUTER在前面已被定义过，则编译下面的命令行:

```
#define INTEGER_SIZE 16
```

否则，编译下面的命令行:

```
#define INTEGER_SIZE 32
```

如果在这组条件编译命令之前曾出现以下命令行:

```
#define COMPUTER 0
```

或将COMPUTER定义为任何字符串，甚至是

```
#define COMPUTER
```

则预编译后程序中的INTEGER_SIZE都用16代替，否则都用32代替。

### 2. 格式2

```
#ifndef 宏名
    程序段1
[#else
    程序段2]
#endif
```

同样，方括号里的内容可缺省。其功能为: 如果宏名没被定义过，则编译程序段1，否则编译程序段2。这种形式与第一种形式的作用相反。

### 3. 格式3

```
#if 表达式
    程序段1
[#else
    程序段2]
#endif
```

表达式是一个常量表达式，其功能为: 如果表达式的值不为零，则编译程序段1; 否则，编译程序段2。

各种条件编译格式可以嵌套使用。例如:

```
#if X>4
    #if X>4&&X<8
      int x=5;
    #else
      int x=2;
    #endif
#else
     int x=100;
#endif
```

**例6.4** 输入一行字母字符，根据需要设置条件编译，使之能将字母全改成大写输出或全改成小写输出。

```
#include <iostream.h>  // Li0604.cpp
#define LETTER 0
void main( )
{  char str[20]="C Language",c;
   int i=0;
```

```
    while ((c=str[i])!='\0')
    {     i++;
          #if LETTER
                if (c>='a'&& c<='z')
                      c=c-32;
          #else
                if (c>='A'&&c<='Z')
                      c=c+32;
          #endif
          cout<<c;
    }
    cout<<endl;
}
```

例6.5　在调试程序时，常常在源程序中插入一些专门为调试程序用的语句，常用的语句有输出语句等，其目的是为了监测程序的执行情况。当然也可以采用系统提供的设置断点和监视表达式等方法。在使用插入语句的方法时，调试完成后还需要逐一删除，这样很麻烦。可以使用条件编译命令使得在调试时插入的语句不用删除。其方法是：使用条件编译命令在调试时使专用于调试的程序段参与编译，调试结束后，重新编译时，使用于调试的程序段不参与编译。例如：

```
#define DEBUG 1
    .
    .
    .
#if DEBUG==1
          count<<"a="<<a<<"b="<<b<<endl;
#endif
    .
    .
    .
```

在该例中，#if与#endif之间的程序段是调试时参加编译的程序段。当调试完成后，将

```
#define DEBUG 1
```

改为：

```
#define DEBUG 0
```

重新编译时，用于调试的程序段将不参与编译。

## 习题6

1. 编写一个程序，将求两个实数的最小值的函数放在一个头文件中，在源程序文件中包含该头文件，并实现输入3个实数，求出最小值。
2. 输入两个整数，求它们的乘积。用带参的宏实现。
3. 用带参的宏，从3个数中找出最大数。
4. 求圆面积。用带参的宏实现。
5. 用条件编译方法实现以下功能：输入一行电报文字，可以任选两种输出，一种为原文输出；另一种为将字母变成其后第五个字母（如‘a’变成‘e’，……，‘z’变成‘d’。其他字符不变。）。用#define命令来控制是否要译成密码。例如：

```
#define CHANGE 1
```

则输出密码。若

```
#define CHANGE 0
```

则不译成密码，按原码输出。

# 第7章 数 组

## 7.1 数组的定义及应用

C++除了提供前面介绍的基本数据类型外，还提供了构造数据类型，来满足不同应用的需要。构造数据类型是由基本类型数据按一定规则组成的，因此有些书中把它们称做“导出类型”。构造数据类型包括数组、结构体、共用体和类。

本章只介绍数组构造类型。数组是有序数据的集合，数组的每一个元素都属于同一种数据类型，用一个统一的数组名和下标来唯一地确定数组中的元素。下面介绍C++语言中是如何定义和使用数组的。

### 7.1.1 一维数组的定义及使用

#### 1. 一维数组的定义

一维数组的定义方式为:

```
<类型说明符> <数组名>[<常量表达式>];
```

例如:

```
int array[10];
```

它表示数组名为array，此数组中有10个元素。

说明:

(1) 数组名命名规则和变量名相同，遵循标识符命名规则。

(2) 数组名后是用方括号括起来的常量表达式，不能用圆括号，如:

```
int array(10);
```

它表示定义一个int型变量array，其初值是10，而不是定义数组。

(3) 常量表达式表示元素的个数，即数组长度。例如，在array[10]中，10表示array数组中有10个元素，下标从0开始，这10个元素是: array[0]，array[1]，array[2]，…，array[8]，array[9]。注意不能使用数组元素array[10]。

(4) 常量表达式中可以包括常量和符号常量，不能包含变量。即C++不允许对数组的大小做动态定义。例如:

```
int n;
cin>>n;
int array[n];
```

是不行的。

如果想方便地修改数组的大小，则可以使用符号常量或const型变量。例如:

```
#define N 100
const int SIZE=200;
int array[N];
float x[SIZE];
```

#### 2. 一维数组元素的引用

数组必须先定义，然后使用。C++语言规定只能逐个引用数组元素而不能一次引用整个数组。数组

元素的表示形式为:

数组名[<下标>]

<下标>可以是整型常量或整型表达式。例如:

array[8]、array[2*3]、array[i]（i为整型变量）均是合法的引用。

**例7.1** 数组元素的引用。

```
#include <iostream.h> // Li0701.cpp
#include <iomanip.h>
void main( )
{
    int i, a[10];
    for (i=0; i<=9; i++)
          a[i]=i;
    for (i=9; i>=0; i--)
          cout<<setw(4)<<a[i];
    cout<<endl;
}
```

程序运行结果如下:

```
   9   8   7   6   5   4   3   2   1   0
```

程序使a[0]到a[9]的值为0~9，然后按逆序输出。

### 3. 一维数组的初始化

在引用数组元素时，所引用的数组元素必须有确定的值。对数组元素的初始化可以用下列方法实现:

（1）在定义数组时对数组元素赋予初值。例如:

```
int a[10]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

将数组元素的初值依次放在一对花括号内，用逗号隔开。经过上面的定义和初始化之后，a[0]=0，a[1]=1，a[2]=2，a[3]=3，a[4]=4，a[5]=5，a[6]=6，a[7]=7，a[8]=8，a[9]=9。

（2）可以给一部分元素赋初值。例如:

```
int a[10]={0, 1, 2, 3};
```

定义a数组有10个元素，但花括号内只提供4个初值，这表示只给前面4个元素赋初值，后6个元素值为0。

（3）如果想使一个数组中的全部元素值均为0，可以写成:

```
int a[10]={0, 0, 0, 0, 0, 0, 0, 0, 0, 0}; 或 int a[10]={0};
```

但不能写成: int a[10]={0*10};

请读者思考: int a[10]={5}; 数组中各个元素的值是多少?

（4）在给全部数组元素赋初值时，可以不指定数组长度。例如:

```
int a[ ]={0, 1, 2, 3, 4};
```

花括号中列举了5个值，因此C++编译器认为数组a的元素个数为5。注意: 如果要定义的数组大小比列举的数组的初值个数大时，则必须说明数组的大小。

（5）如果把数组定义为全局数组或局部静态数组时，C++编译器自动地将所有数组元素的初值置为0。如果把数组定义为局部动态数组时，数组元素无确定的初值。

**例7.2** 输出数组为全局变量、静态变量及局部变量时的初值。

```
#include <iostream.h>          // Li0702.cpp
int x[5];                      // 全局数组，静态型，各元素初值为0
void main( )
```

```
{   static int y[5];              //局部数组，静态型，各元素初值为0
    int z[5];                     //局部数组，动态型，各元素初值不确定
    for(int i=0; i<5; i++)        cout<<x[i]<<'\t';
    cout<<endl;
    for(i=0; i<5; i++) cout<<y[i]<<'\t';
    cout<<endl;
    for(i=0; i<5; i++) cout<<z[i]<<'\t';
    cout<<endl;
}
```

执行该程序时，输出的第一行为5个0，第二行也是5个0，第三行是随机值，即每次执行该程序时，输出的值可能是不同的。

### 4. 一维数组程序举例

**例7.3** 用数组求Fibonacci数列的前20项和前20项之和。

```
#include <iostream.h> // Li0703.cpp
void main( )
{   int i, f[20]={1, 1}, sum=f[0]+f[1];
    for (i=2; i<20; i++)
    {     f[i]=f[i-2]+f[i-1];
          sum+=f[i];
    }
    for (i=0; i<20; i++)
    {     cout<<f[i]<<'\t';
          if ((i+1)%5==0) cout<<'\n';
    }
    cout<<"前20项和为: "<<sum<<'\n';
}
```

运行情况如下:

```
1       1       2       3       5
8       13      21      34      55
89      144     233     377     610
987     1597    2584    4181    6765
```

前20项和为: 17710

**例7.4** 将一个数分解到数组中，然后正向输出，反向输出。

```
#include <iostream.h>  //Li0704.cpp
void main( )
{   int i, j=0, k, a[20];
    cout<<"请输入一个整数:";
    cin>>i;
    k=i;
    while(k>0)          //将一个数分解到数组a中
    {     a[j++]=k%10;
          k=k/10;
    }
    cout<<"正向输出序列:"<<endl;
    for(k=j-1; k>=0; k--) cout<<a[k]<<'\t';
    cout<<endl;
    cout<<"反向输出序列:"<<endl;
```

```
    for(k=0; k<j; k++) cout<<a[k]<<'\t';
    cout<<endl;
}
```

运行情况如下:

```
请输入一个整数:3265<Enter>
正向输出序列:
3     2     6     5
反向输出序列:
5     6     2     3
```

### 7.1.2 一维数组做函数参数

在定义函数时，除用简单变量作为函数的形参外，还可以用一维数组作为函数的形参。在函数调用时，相应地用数组元素及数组名作为函数的实参。

#### 1. 一维数组元素做函数的实参

由于实参可以是表达式，数组元素可以是表达式的组成部分，因此数组元素也可以做函数的实参，与变量做实参一样，是单向值传递。

**例7.5** 有两个数组a和b，各有n个元素，将它们对应地逐个比较，即a[0]与b[0]比，a[1]与b[1]比……如果a数组中的元素大于b数组中的对应元素的个数多于b数组中的元素大于a数组中对应元素的个数（例如，a[i]大于b[i] 5次，b[i]大于a[i] 2次，其中i每次为不同的值），则认为a数组大于b数组，并分别统计出两个数组对应元素大于、等于、小于的次数。

```
#include <iostream.h>   // Li0705.cpp
void main( )
{  int large(int, int);
   int a[10], b[10], i, j=0, k=0, m=0, n;
                        // a[i] > b[i]个数为j, a[i] < b[i]个数为m, a[i] == b[i]个数为k
   cout<<"请输入数组元素个数:\n";
   cin>>n;
   cout<<"输入数组a:\n";
   for (i=0; i<n; i++)
         cin>>a[i];
   cout<<"输入数组b:\n";
   for (i=0; i<n; i++) cin>>b[i];
   for (i=0; i<n; i++)
   {      if (large(a[i], b[i])==1) j=j+1;
          else if ( large(a[i], b[i])==0 ) k=k+1;
          else m=m+1;
   }
   cout<<" a[i]大于b[i]的次数 "<<j<<endl;
   cout<<" a[i]等于b[i]的次数 "<<k<<endl;
   cout<<" a[i]小于b[i]的次数 "<<m<<endl;
   if (j>m)
          cout<<"数组 a 大于数组 b\n";
   else if (j<m)
          cout<<"数组 a 小于数组 b\n";
   else cout<<"数组 a 等于数组 b\n";
}
int large (int x, int y)
{   int flag;
```

```
    if (x>y)  flag=1;
    else if (x<y)  flag=-1;
    else flag=0;
    return(flag);
}
```

运行情况如下:

```
请输入数组元素个数:
10<Enter>
输入数组a:
1 5 4 8 7 6 3 2 -10 9<Enter>
输入数组b:
-5 3 10 -3 7 9 1 -1 2 6<Enter>
 a[i]大于b[i]的次数 6
 a[i]等于b[i]的次数 1
 a[i]小于b[i]的次数 3
数组 a 大于数组 b
```

### 2. 数组名做函数的实参

可以用数组名做函数参数，此时实参与形参都应使用数组名。

**例7.6** 将一个一维数组中的值按逆序重新存放。例如，原来的顺序为7，2，5，4，3，6，1。要求改为1，6，3，4，5，2，7。

逆序重新存放的思路是：第一个元素与最后一个元素对调，这里为7与1对调；第二个元素与倒数第二个元素对调，这里为2与6对调；依此类推……直到中间那个元素为止。

```
#include <iostream.h> // Li0706.cpp
#define N 7
void inverse(int b[N])
{  int i;
   for(i=0; i<N/2; i++)
   {     int t=b[i];
         b[i]=b[N-1-i];
         b[N-1-i]=t;
   }
}
void main( )
{  int a[N]={ 7, 2, 5, 4, 3, 6, 1};
   cout<<"原始数组元素为:"<<endl;
   for(int i=0; i<N; i++)
         cout<<a[i]<<'\t';
   cout<<'\n';
   inverse(a);
   cout<<"逆序后的数组元素为:"<<endl;
   for(i=0; i<N; i++)
         cout<<a[i]<<'\t';
   cout<<'\n';
}
```

程序运行结果为:

```
原始数组元素为:
7       2       5       4       3       6       1
```

```
逆序后的数组元素为:
1       6       3       4       5       2       7
```

说明:

(1) 用数组名做函数参数，应该在主调函数和被调函数中分别定义数组，上例中a既是形参数组名，又是实参数组名，应分别在其所在函数中定义，不能只在一方定义。

(2) 实参数组与形参数组类型应一致，这里均为int型，如不一致，结果将出错。

(3) 为了编写通用程序，形参数组也可以不指定大小，可在定义数组时在数组名后面跟一个空的方括号，另设一个参数，传递数组元素的个数，例7.6可以改写为如下形式:

```
#include <iostream.h>                    //Li0706_1.cpp
#define N 7
void inverse(int b[], int n)             //这样定义具有通用性
{     int i;
      for(i=0; i<n/2; i++)
      {     int t=b[i];
            b[i]=b[n-1-i];
            b[n-1-i]=t;
      }
}
void main( )
{     int a[N]={ 7, 2, 5, 4, 3, 6, 1};
      cout<<"原始数组元素为:"<<endl;
      for(int i=0; i<N; i++)
            cout<<a[i]<<'\t';
      cout<<'\n';
      inverse(a, N);
      cout<<"逆序后的数组元素为:"<<endl;
      for(i=0; i<N; i++)
            cout<<a[i]<<'\t';
      cout<<'\n';
}
```

(4) 用数组名做函数实参时，不是把数组的值传递给形参，而是把实参数组的起始地址（“起始地址”也称为“首地址”）传递给形参数组名，这样两个数组就共占同一段内存单元，见图7-1。

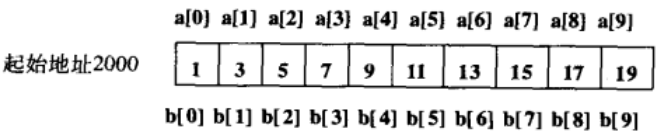



图7-1　数组名做参数的意义

若a的起始地址为2000，则b数组的起始地址也是2000，显然，a和b同占一段内存单元，a[0]与b[0]同占一个单元……由此可以看到，形参数组中各元素的值如发生变化会使实参数组元素的值同时发生变化，从图7-1中看是很容易理解的。这一点与变量做函数参数的情况不同，请注意。在程序设计中可以有意识地利用这一特点改变实参数组元素的值，如上例的逆序。

**例7.7**　用冒泡法对6个数排序（由小到大）。

冒泡法的思想是：将相邻两个数比较，将小的调到前头，见图7-2。

若有6个数。第1次比较8和4，对调，第2次比较8和9，不对调，第3次比较9和6，对调……如此共进行5次，得到4-8-6-5-2-9的顺序，可以看到：最大的数9已经沉到底部，成为最下面的数，而小的数逐步

上升。最小的数2已经向上浮起一个位置。经第1趟比较及交换后，已得到最大的数。然后进行第2趟比较，对余下的前5个数按上面的方法进行比较，见图7-3。经过4次比较，得到次大的数8。依此类推，6个数要经过5趟比较及交换才能使6个数按大小顺序排列。在第1趟中要进行两个数的比较共5次，在第2趟中比较4次……第5趟比较1次。如果有n个数，则要进行n－1趟比较。在第1趟比较中要进行n－1次两两比较，在第j趟比较中要进行n-j次两两比较。

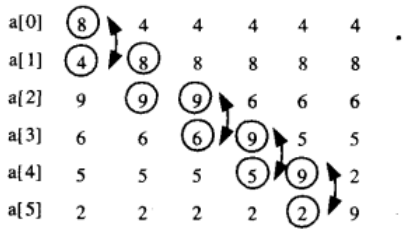



图7-2 冒泡法第一趟比较及交换的过程

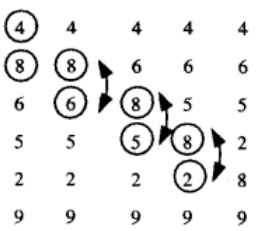



图7-3 冒泡法第二趟比较及交换的过程

```
#include <iostream.h> // Li0707.cpp
#define N 6
void bubble_sort(int a[], int n)
{   int i, j, t;
    for(i=0; i<n-1; i++)
          for(j=0; j<n-1-i; j++)
               if(a[j]>a[j+1])
               {     t=a[j];  a[j]=a[j+1];  a[j+1]=t; }
}
void main( )
{   int a[N];
    int i;
    cout<<"请输入"<<N<<"个数:"<<endl;
    for(i=0; i<N; i++)
          cin>>a[i];
    bubble_sort(a, N);
    cout<<"排好序的数为:"<<endl;
    for(i=0; i<N; i++) cout<<a[i]<<'\t';
    cout<<endl;
}
```

程序运行情况如下:

```
请输入6个数:
8  4  9  6  5  2<Enter>
排好序的数为:
2        4        5        6        8        9
```

从上面的分析中可以看出冒泡法数组元素交换太频繁，有没有更好的方法进行排序呢？这里介绍一种较好的排序方法，即选择法排序。

**例7.8** 用选择法对数组中的6个数排序（从小到大）。

假定6个元素存放在a[0]……a[5]中。

选择法的思想是：第1趟扫描，将a[0]到a[5]中最小数的下标找到，设为p，若 p!=0，则a[0]与a[p]交换位置；第2趟扫描，将a[1]到a[5]中最小数的下标找到，设为p，若 p!=1，则a[1]与a[p]交换位置……依此类推，每扫描一次找出一个未经排序的数中最小的一个，共需5次扫描，第i次扫描时（i=0~4），扫描

范围的第1个元素的下标为i，最后一个元素的下标为5。

选择法的步骤：

a[0]a[1]a[2]a[3]a[4]a[5]

8 4 9 6 5 2 未排好序的数据

2 4 9 6 5 8 将6个数中最小的数与a[0]交换

2 4 9 6 5 8 余下的5个数中最小的数即a[1]，不交换

2 4 5 6 9 8 将余下的4个数中最小的数与a[2]交换

2 4 5 6 9 8 余下的3个数中最小的数与a[3]，不交换

2 4 5 6 8 9 将余下的2个数中最小的数与a[4]交换，至此完成排序

```
#include <iostream.h>                                    //Li0708.cpp
#define N 6
void select_sort(int a[], int n)
{  int i,j,p,t;
   for(i=0; i<n-1; i++)
   {     p=i;
         for(j=i+1; j<n; j++)
               if(a[j]<a[p]) p=j;                        //记下每一趟最小元素的下标
         if(p!=i)
         {     t=a[p]; a[p]=a[i]; a[i]=t;       }
   }
}
void main( )
{  int a[N];
   int i;
   cout<<"请输入"<<N<<"个数:"<<endl;
   for(i=0; i<N; i++) cin>>a[i];
   select_sort(a, N);
   cout<<"排好序的数为:"<<endl;
   for(i=0; i<N; i++) cout<<a[i]<<'\t';
   cout<<endl;
}
```

程序运行情况如下：

```
请输入6个数:
8  4  9  6  5  2<Enter>
排好序的数为:
2        4        5        6        8        9
```

选择法排序的另一个实现方法为：将第0个元素与第1个元素进行比较，若第0个元素大于第1个元素，则两个数进行交换；否则不交换。再将第0个元素与第2个元素进行比较，若第0个元素大于第2个元素，则两个数进行交换……依次类推，直到第0个元素与最后一个元素进行比较，若大于最后一个元素，则两个数交换。这时，已将数组中最小的数移动到第0个元素的位置。再从第1个元素开始，用同样的方法，找出次小的数移动到第1个元素的位置。依次类推，直到将次大的数移动到第4个元素位置，这时第5个元素（最后一个元素）就是最大数，排序至此结束。

```
#include <iostream.h>                    //Li0708_1.cpp 选择法排序的变种
#define N 6
void select_sort(int a[], int n)
{  int i , j, t;
   for(i=0; i<n-1; i++)                  //按升序排序
```

```
        for(j=i; j<n; j++)
            if(a[i]>a[j])
            {       t=a[i];  a[i]=a[j];  a[j]=t;  }
}
void main( )
{   int a[N];
    int i;
    cout<<"请输入"<<N<<"个数:"<<endl;
    for(i=0; i<N; i++) cin>>a[i];
    select_sort(a, N);
    cout<<"排好序的数为:"<<endl;
    for(i=0; i<N; i++) cout<<a[i]<<'\t';
    cout<<endl;
}
```

运行情况同上。

下面再介绍一种排序方法——插入法排序，插入法排序的方法分为前插和后插两种，分别介绍之。

**例7.9** 插入法排序。后插算法为：设数组有N个元素，第i次循环结束后，数组前i个元素排成升序，即有a[0]≤a[1]≤...≤a[i－1]。现在要将a[i]元素插入，使前i+1个元素保持升序（如图7-4所示）。首先将a[i]送p，然后将p依次与a[i－1]，a[i－2]，...，a[0]进行比较，将比p大的元素依次右移一个位置，直到发现某个j（0≤j≤i－1），有a[j]≤p成立，则使得a[j+1]=p；如果不存在这样的a[j]，那么在比较过程中，a[i－1]，a[i－2]，...，a[0]都依次右移一个位置，此时使得a[0]= p。

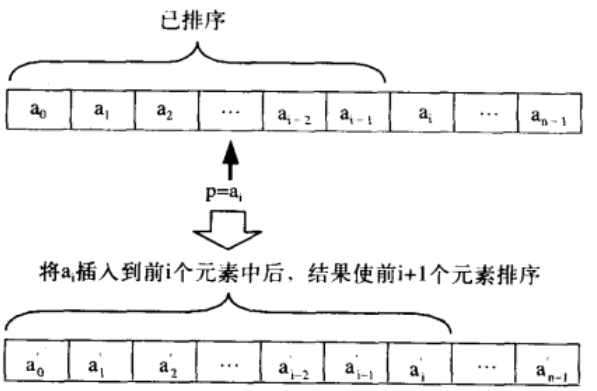



图7-4 插入法排序示意图

```
#include <iostream.h>                          // Li0709.cpp 直接插入排序后插算法
#define N 6
void ba_ins_sort(int a[], int n)
{     int i, j, p;
      for(i=1; i<n; i++)
      {     p=a[i];
            for(j=i-1; j>=0&&p<a[j]; j--)// 将比p大的元素依次右移一个位置
                        a[j+1]=a[j];
            a[j+1]=p;
      }
}
void main( )
{     int a[N], i;
```

```
    cout<<"请输入"<<N<<"个数:"<<endl;
    for(i=0; i<N; i++) cin>>a[i];
    ba_ins_sort(a, N);
    cout<<"排好序的数为:"<<endl;
    for(i=0; i<N; i++) cout<<a[i]<<'\t';
    cout<<endl;
}
```

运行情况同例7.8。

插入法排序的另一实现方法为前插算法：设数组有N个元素，第i次循环结束后，数组前i个元素排成升序，即有a[0]≤a[1]≤...≤a[i-1]。现在要将a[i]元素插入，使前i+1个元素保持升序。首先将a[i]送p，然后将p依次与a[0], a[1], ..., a[i-1]进行比较，直到发现某个j（0≤j≤i-1），有a[j]>p成立，则把a[j], a[j+1], …, a[i-1]依次右移一个位置，使得a[j]=p。

```
#include <iostream.h>                          //Li0709_1.cpp 直接插入排序前插算法
#define N 6
void ah_ins_sort(int a[], int n)
{  int i, j, p;
   for(i=1; i<n; i++)
   {     p=a[i];
         for(j=0; j<i&&p>=a[j]; j++);          //找到待插位置
         for(int k=i; k>j; k--)                //将比p大的元素依次右移一个位置
               a[k]=a[k-1];
         a[j]=p;
   }

}
void main( )
{  int a[N], i;
   cout<<"请输入"<<N<<"个数:"<<endl;
   for(i=0; i<N; i++) cin>>a[i];
   ah_ins_sort(a, N);
   cout<<"排好序的数为:"<<endl;
   for(i=0; i<N; i++) cout<<a[i]<<'\t';
   cout<<endl;
}
```

运行情况同例7.8。

**例7.10**　用筛选法求1~100之间的素数。所谓“筛选法”指的是，从1开始逐个判断它们是否为素数，如果为非素数，就去掉，最后剩下的就是素数。算法为：用一个一维数组a存放2，3，4，5，…，98，99，100；从a[0]开始，其后的各个元素是a[0]的倍数，则该数不是素数，将该元素值置为0；再从a[1]开始，其后的各个元素是a[1]的倍数，则该数不是素数，将该元素值置为0；依此类推，直至从a[99] 开始，其后的各个元素是a[99]的倍数，则该数不是素数，将该元素值置为0。这时数组a中不为0的元素，均为素数。要求每行输出5个素数。

```
#include <iostream.h>          //Li0710.cpp
#define N 100
void prime(int a[],int n)
{  int i, j;
   for(i=1; i<n-1; i++)        //a[0]（即1）不是素数，因此从a[1]开始判断
         for(j=i+1; j<n; j++)
```

```
                if(a[i]!=0&&a[j]!=0)
                        if(a[j]%a[i]==0) a[j]=0;

}
void main( )
{   int a[N], i, n;
    for(i=0; i<N; i++) a[i]=i+1;
    prime(a, N);
    cout<<"1~100内的素数为:"<<endl;
    for(i=1, n=0; i<N; i++)
    {
            if(a[i]!=0)
            {       cout<<a[i]<<'\t';
                    n++;
                    if(n%5==0) cout<<'\n';
            }
    }
}
```

运行情况如下:

```
1~100内的素数为:
2       3       5       7       11
13      17      19      23      29
31      37      41      43      47
53      59      61      67      71
73      79      83      89      97
```

**例7.11** 采用顺序查找法，在长度为n的一维数组中查找值为x的元素，若找到，则返回数组元素的下标；若找不到则返回－1。即从数组的第一个元素开始，逐个与被查值x进行比较。要求用一个函数实现对数组元素的顺序查找。

```
#include <iostream.h> //Li0711.cpp 顺序查找
#define M 10
int search(int a[ ], int x, int n)
{    int i;
     for(i=0; i<n; i++)
            if(x==a[i]) return i;
return -1;
}
void main( )
{   int array[ ]={6, 3, 18, 24, 9, 32, 6, 46, 1, 12}, i, p, x;
    cout<<"输入要查找的元素:";
    cin>>x;
    p=search(array, x, M);
    cout<<"原始数组元素为:"<<endl;
    for(i=0; i<M; i++)
          cout<<array[i]<<'\t';
    if(p>=0) cout<<"查找成功! 数组元素的下标为:"<<p<<endl;
    else cout<<"未查找到! "<<endl;
}
```

运行情况如下:

```
输入要查找的元素:9<Enter>
原始数组元素为:
6        3        18       24       9        32       6        46       1        12
查找成功! 数组元素的下标为:4
```

**例7.12**　采用折半查找法，在长度为n的一维数组中查找值为x的元素。折半查找的前提是：数组中的元素已经排序（这里假定是非递减排序）。算法为：将x与数组的中间项进行比较，若被查元素x等于数组中间项的值，则查找成功，结束查找；若被查元素x小于数组中间项的值，则取中间项以前的部分以相同的方法进行查找；若被查元素x大于数组中间项的值，则取中间项以后的部分以相同的方法进行查找；这个过程一直进行到查找成功或数组中的元素个数为0（即查找失败）为止。要求用一个函数实现对数组元素的折半查找。

```
#include <iostream.h>      // Li0712.cpp 折半查找
#define M 10
int bi_search(int a[ ], int x, int n)
{   int low=0, mid, up=n-1;
    while (low<=up)
    {       mid=(low+up)/2;
            if(x==a[mid])
                 return mid;
    else if(x<a[mid])
          up=mid-1;
    else
          low=mid+1;
    }
    return -1;
}
void main( )
{   int array[ ]={1, 3, 6, 24, 30, 32, 36, 46, 100, 120}, i, p, x;
    cout<<"输入要查找的元素:";
    cin>>x;
    p=bi_search(array, x, M);
    cout<<"原始数组元素为:"<<endl;
    for(i=0; i<M; i++)
          cout<<array[i]<<'\t';
    if(p>=0) cout<<"查找成功! 数组元素的下标为"<<p<<endl;
    else cout<<"未查找到! "<<endl;
}
```

运行情况如下：

```
输入要查找的元素:6<Enter>
原始数组元素为:
1     3     6     24     30     32     36     46     100     120
查找成功! 数组元素的下标为2
输入要查找的元素:48 ↙
原始数组元素为:
1     3     6     24     30     32     36     46     100     120
未查找到!
```

**例7.13**　求两个集合的交集，并求出交集的元素个数。例如：集合$A$={4, 8, 2, 1, 9, 10}，集合$B$={2, 5, 3, 9, 7}，两个集合的交集为$C$={2, 9}，交集中的元素个数为2。

```
#include <iostream.h>                                    //Li0713.cpp
int search(int b[ ], int x, int n)                       //例7.11中编写的函数
{   int i;
    for(i=0; i<n; i++)
          if(x==b[i]) return i;
    return -1;
}
int intersection(int a[ ], int b[ ], int c[ ], int m, int n)
{   int i, j, k=0;
    for(i=0; i<m; i++)
          if((j=search(b, a[i], n))!=-1) c[k++]=b[j]; //用数组元素作为函数的实参
    return k;
}
void main( )
{  int a[ ]={4, 8, 2, 1, 9, 10},b[ ]={2, 5, 3, 9, 7}, c[20], count, i;
   count=intersection(a, b, c, 6, 5);                    //用数组名作为函数的实参
   cout<<"数组a和数组b的交集:";
   for(i=0; i<count; i++)
         cout<<c[i]<<'\t';
   cout<<endl;
   cout<<"数组a和数组b交集的元素个数:"<<count<<endl;
}
```

运行情况如下:

```
数组a和数组b的交集:2   9
数组a和数组b交集的元素个数:2
```

### 7.1.3 多维数组的定义及使用

具有两个或两个以上下标的数组称为多维数组。下面以二维数组为例说明多维数组的定义及使用方法。

#### 1. 二维数组的定义

二维数组定义的一般形式为:

```
<类型说明符> <数组名>[<常量表达式1>][<常量表达式2>];
```

例如:

```
int a[3][4];
```

定义a数组为3×4(3行4列)的数组,a数组中的数组元素为整型数。

我们可以把二维数组看做是一种特殊的一维数组,它的元素又是一个一维数组。例如可以把a数组看做是一个一维数组,它有3个元素a[0]、a[1]、a[2],每个元素又是一个包含4个元素的一维数组,见图7-5。可以把a[0]、a[1]、a[2]看做是3个一维数组的名字。

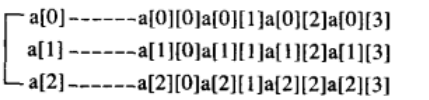



图7-5 二维数组元素

此处把a[0]、a[1]、a[2]看做一维数组名。C++语言的这种处理方式在数组初始化和用指针表示时显得很方便,读者在以后的使用中会有所体会。

C++语言中,二维数组中的元素在内存中是按行存放的,即在内存中先顺序存放第一行的元素,再存放第二行的元素……依此类推。图7-6是a[3][4]数组在内存的存放情况。

C++允许使用多维数组,对数组的维数没有限制。有了二维数组的基础,再掌握多维数组是不困难的。例如,定义三维数组的方法是:

```
int a[2][3][4];
```

可将其理解为：由2个二维数组构成，每个二维数组又是由3个一维数组构成的。

多维数组元素在内存中的存放顺序：第一维的下标变化最慢，最右边的下标变化最快。例如，上述三维数组的元素存放顺序为：

a[0][0][0]→a[0][0][1]→a[0][0][2]→a[0][0][3]→
a[0][1][0]→a[0][1][1]→a[0][1][2]→a[0][1][3]→
a[0][2][0]→a[0][2][1]→a[0][2][2]→a[0][2][3]→
a[1][0][0]→a[1][0][1]→a[1][0][2]→a[1][0][3]→
a[1][1][0]→a[1][1][1]→a[1][1][2]→a[1][1][3]→
a[1][2][0]→a[1][2][1]→a[1][2][2]→a[1][2][3]

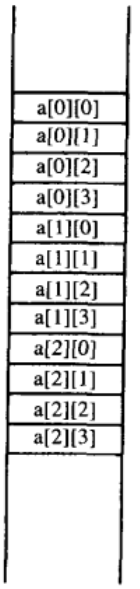



图7-6　二维数组元素的存储

### 2. 二维数组的引用

二维数组的元素的引用形式为：

```
数组名[<下标1>][<下标2>]
```

下标必须是整型表达式，如a[1][2], a[2 - 1][2*2 - 1], a[i - 1][2*j]（i、j为整型变量）均是合法的引用。

使用数组元素就像使用一个简单变量一样，包括输入／输出、计算等。它可以出现在表达式中，也可以被赋值，例如：

```
a[1][2]=a[2][3]/2
```

在使用数组元素时，要注意下标值应该在已定义的数组大小的范围内。常出现的错误是：

```
int a[3][4];
...
a[3][4]=5;
```

定义a为3×4的数组，它可用的行下标值范围为0~2，列下标值范围为0~3。用a[3][4]超出了数组的范围。

请读者严格区分在定义数组时用a[3][4]和引用元素时的a[3][4]的区别。前者a[3][4]中的3和4是用来定义数组各维的大小，后者a[3][4]中的3和4是下标值，a[3][4]代表某一个元素。

### 3. 二维数组的初始化

可以用下面的方法对二维数组初始化：

（1）分行给二维数组赋初值。如：

```
int a[3][4]={{1, 2, 3, 4},{5, 6, 7, 8},{9, 10, 11, 12}};
```

这种赋初值方式比较直观，把第1个花括号内的数据给第1行的元素，把第2个花括号内的数据给第2行的元素……即按行赋初值。

（2）可以将所有数据写在一个花括号内，按数组排列的顺序对各元素赋初值。如：

```
int a[3][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

效果与第1种方法相同。但以第1种方法为好，按行赋初值，界限清楚。用第2种方法如果数据多，写成一大片，容易遗漏，也不易检查。

（3）可以对部分元素赋初值。如：

```
int a[3][4]={{1}, {3}, {5}};
```

它的作用是对各行第0列的元素赋初值，其余元素值自动为0。赋初值后数组各元素为：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 \end{bmatrix}$$

(4) 可以根据给定的初始化的数据，自动确定数组的行数。如:

```
int a[ ][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

系统会根据数据的总个数分配存储空间，一共12个数据，每行4列，当然可以确定为3行。

又如:

```
int b[ ][4]={1, 2, 3, 4, 5, 6, 7, 8, 9};
```

定义数组b为3行4列的数组，b数组也可以这样来定义:

```
int b[ ][4]={{1, 2, 3, 4}, {5, 6, 7, 8}, {9}};
```

后者的定义方法比前者清晰。注意，定义数组时只能省略行数，不能省略列数。若省略列数，则行列之间的关系就不唯一了。

与一维数组雷同，当定义为静态的多维数组或全局变量的多维数组时，系统自动地将数组的各元素的初值赋为0。数组不能整体赋值，要将一个数组的值赋给另一个数组时，必须逐个元素赋值。如:

```
int a[3][4], b[3][4];
```

要将数组a中的各个元素依次赋给数组b时，不能写成b=a，而必须用下面的循环语句逐个赋值:

```
for(int i=0; i<3; i++)
        for(int j=0; j<4; j++)
             b[i][j]=a[i][j];
```

### 7.1.4 二维数组名用做函数参数

二维数组也可以用做函数参数，同一维数组，二维数组的数组元素和多维数组名均可作为函数参数。

#### 1. 二维数组元素做函数参数

二维数组元素也可以做函数参数，也与变量做实参一样，是单向值传递。

**例7.14** 有两个二维数组a[3][4]和b[3][4]，统计两个数组中对应元素相等的个数。

```
#include <iostream.h>          //Li0714.cpp 二维数组元素做函数参数
void main( )
{   int equal(int, int);       //函数原型说明
    int a[3][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12},
          b[3][4]={12, 2, 3, 4, 10, 6, 7, 11, 9, 5, 8, 1}, i, j, k=0;
    for (i=0; i<3; i++)
          for(j=0; j<4; j++)
                if (equal(a[i][j], b[i][j])==1) k++;
    cout<<"两个数组中对应元素相等的个数为:"<<k<<endl;
}
int equal(int x, int y)
{   int flag;
    if (x==y)  flag=1;
    else flag=0;
    return(flag);
}
```

运行结果为:

```
两个数组中对应元素相等的个数为:6
```

#### 2. 二维数组名做函数参数

二维数组名做函数参数，在被调函数中对形参数组定义时可以指定每一维的大小，也可以省略第一维的大小说明。例如:

```
int a[4][5]; 或int a[ ][5];
```

二者都合法而且等价。但是不能把第二维以及其他高维的大小说明省略。如下面是不合法的:

```
int a[ ][ ]; 或int a[3][ ];
```

因为从实参传递来的是数组的首地址，在内存中数组是按行存放的，并不区分行和列，如果在形参中不说明列数，则系统无法确定行列数。不能只指定第一维而省略第二维。

### 3. 二维数组程序举例

**例7.15** 有一个3×4的二维数组，要求编程序求出其中值最小的那个元素，以及其所在的行号和列号。

算法思想: 首先将数据输入到数组a中，并将数组中的任一元素值如a[0][0]作为最小值min，将min分别与数组中的各个元素比较，若某个元素小于min，则将该元素值赋给min，并记下该元素所在的行号和列号。

```
#include <iostream.h>                    //Li0715.cpp
int min_element(int a[][4], int &row, int &colum)
{  int i, j, min;
   min=a[0][0];
   for (i=0; i<=2; i++)
         for (j=0; j<=3; j++)
               if (a[i][j]<min)
               {      min=a[i][j];      row=i;      colum=j;   }
   return min;
}
void main( )
{  int i, j, row=0, colum=0, min;
   int a[3][4];
   cout<<"请输入3行4列的二维数组:";
   for (i=0; i<=2; i++)                  //输入二维数组
         for (j=0; j<=3; j++)
               cin>>a[i][j];
   min=min_element(a, row, colum);
   cout<<"最小值="<<min<<endl;
   cout<<"所在行号="<<row<<"      所在列号="<<colum<<endl;
}
```

运行情况如下:

```
请输入3行4列的二维数组:1 2 3 4 9 8 7 6 -10 5 2 -5<Enter>
最小值=-10
所在行号=2   所在列号=0
```

**例7.16** 将一个3×3的二维数组的行和列的元素交换（数学上称为矩阵的转置），并存放到另一个数组中。例如: 将a数组的行和列交换后，存入b数组中。

```
#include <iostream.h>                  //Li0716.cpp 矩阵转置
void transpose(int a[][3], int b[][3])
{  for (int i=0; i<3; i++)
         for (int j=0; j<3; j++) b[j][i]=a[i][j];
}
void main( )
{  int a[3][3]={{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
   int b[3][3], i, j;
   cout<<"数组a:"<<endl;
   for (i=0; i<3; i++)                 //输出二维数组
   {      for (j=0; j<3; j++) cout<<a[i][j]<<'\t';
```

```
            cout<<'\n';
        }
        transpose(a, b);                    //行列交换
        cout<<"数组 b:\n";
        for (i=0; i<3; i++)
        {       for(j=0; j<3; j++) cout<<b[i][j]<<'\t';
                cout<<'\n';
        }
}
```

执行程序后，输出：

```
数组a:
1       2       3
4       5       6
7       8       9
数组b:
1       4       7
2       5       8
3       6       9
```

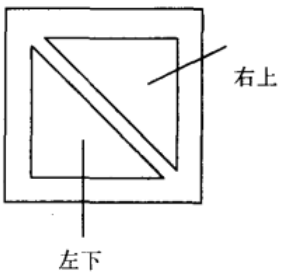



图7-7 矩阵“左下/右上”元素示意

如果在a数组中进行转置，只要将a数组的左下三角形和右上三角形内的元素对调即可，如图7-7所示。

```
// Li0716_1.cpp 在一个数组中进行转置
#include <iostream.h>
void transpose(int a[][3])
{   for (int i=0; i<3; i++)
            for (int j=i+1; j<3; j++)
//注意j的初值不是0，对右上三角形内的元素循环
      {     int t=a[i][j]; a[i][j]=a[j][i];       a[j][i]=t;      }
}
void main( )
{   int a[3][3]={{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int i, j;
    cout<<"原始数组a:"<<endl;
    for (i=0; i<3; i++)
    {       for (j=0; j<3; j++)  cout<<a[i][j]<<'\t';
            cout<<'\n';
    }
    transpose(a);
    cout<<"转置过的数组 a:\n";
    for (i=0; i<3; i++)
    {       for(j=0; j<3; j++) cout<<a[i][j]<<'\t';
            cout<<'\n';
    }
}
```

请读者自己考虑，对左下三角形元素循环，如何实现？

**例7.17** 有如下两个矩阵，求这两个矩阵的乘积$C=AB$（c[i][j]=$\sum_{k=0}^{n-1} a[i][k]*b[k][j]$）。要求用一个函数实现矩阵相乘。

$$A=\begin{bmatrix}1 & 3 & 5\\ 2 & 4 & 6\\ 15 & 7 & 4\\ -2 & 8 & 9\end{bmatrix}\qquad B=\begin{bmatrix}3 & 6 & 2 & 1 & 7\\ 9 & 1 & 3 & -1 & 5\\ 2 & 5 & 8 & 1 & 9\end{bmatrix}$$

```
#include <iostream.h>  //Li0717.cpp矩阵相乘
#define M 4
#define N 3
#define P 5
void main( )
{  void mul(int a[ ][N], int b[ ][P], int c[ ][P]);
   int a[M][N]={{1, 3, 5}, {2, 4, 6}, {15, 7, 4}, {-2, 8, 9}},
       b[N][P]={{3, 6, 2, 1, 7}, {9, 1, 3, -1, 5}, {2, 5, 8, 1, 9}}, c[M][P], i, j;
   mul(a, b, c);
   cout<<"矩阵A为:"<<endl;
   for(i=0; i<M; i++)
   {     for(j=0; j<N; j++) cout<<a[i][j]<<'\t';
         cout<<'\n';
   }
   cout<<"矩阵B为:"<<endl;
   for(i=0; i<N; i++)
   {     for(j=0; j<P; j++) cout<<b[i][j]<<'\t';
         cout<<'\n';
   }
   cout<<"相乘后的矩阵C为:"<<endl;
   for(i=0; i<M; i++)
   {     for(j=0; j<P; j++) cout<<c[i][j]<<'\t';
         cout<<'\n';
   }
}
void mul(int a[ ][N], int b[ ][P], int c[ ][P])
{  int i, j, k;
   for(i=0; i<M; i++)
   for(j=0; j<P; j++)
   {
         c[i][j]=0;
         for(k=0; k<N; k++)
               c[i][j]=c[i][j]+a[i][k]*b[k][j];
   }
}
```

运行结果如下:

```
矩阵A为:
1       3       5
2       4       6
15      7       4
-2      8       9
矩阵B为:
3       6       2       1       7
9       1       3       -1      5
2       5       8       1       9
```

相乘后的矩阵C为：

```
40      34      51      3       67
54      46      64      4       88
116     117     83      12      176
84      41      92      -1      107
```

**例7.18** 计算二维数组a中每一行元素之和，将和放在一个一维数组b中。要求在主函数中初始化二维数组并将每个元素输出，然后调用子函数，分别计算每一行元素之和，在主函数中输出每一行元素之和。

```
#include <iostream.h> //Li0718.cpp
void rsum(int a[ ][4], int row, int b[ ])
{   for(int i=0; i<row; i++)
    {     b[i]=0;
          for(int j=0; j<4; j++)
                b[i]+=a[i][j];
    }
}
void main( )
{   int a[ ][4]={{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}}, b[3];
    for(int i=0; i<3; i++)
    {     for(int j=0; j<4; j++)
                cout<<a[i][j]<<'\t';
          cout<<endl;
    }
    rsum(a, 3, b);
    for(i=0; i<3; i++)
    {     cout<<"第 "<<i<<" 行元素之和是: "<<b[i]<<endl; }
}
```

程序运行结果为：

```
1       2       3       4
5       6       7       8
9       10      11      12
第 0 行元素之和是: 10
第 1 行元素之和是: 26
第 2 行元素之和是: 42
```

## 7.2 字符数组的定义及应用

字符数组可以与前面介绍的数组一样来使用，所不同的是，数组中的每一个元素存放的均为一个字符。

### 7.2.1 字符数组的定义

字符数组的定义与一般数组的定义类似。例如：

```
char c[15];
```

其元素为c[0]~c[14]，每个元素都是字符型变量。

在C++中，可以将字符型数据作为整型数据来处理，整型数据也可以作为字符型数据来处理，注意整型数的值应该在0~255内。从这个意义上来说，字符型和整型之间是通用的，但两者又是有区别的。例如：

```
char c1[10]; int c2[10];
```

为数组c1分配的存储空间是10个字节，而为数组c2分配的存储空间为40个字节。

### 7.2.2 字符数组的初始化

对字符数组初始化，最容易理解的方式是逐个将字符赋给数组中的各元素。如:

```
char c[15]={ 'I', ' ', 'a', 'm', ' ', 'a', ' ', 's', 't', 'u', 'd', 'e', 'n', 't', '.'};
```

把括号中的15个字符常数分别赋给c[0]到c[14] 15个元素。

如果花括号内提供的初值个数（即字符个数）大于数组长度，则按语法错误处理。如果初值个数小于数组长度，则只将这些字符赋给数组中前面那些元素，其余的元素由系统自动赋值为空字符（即'\0'，其ASCII码值为0）。如:

```
char c[15]={ 'H', 'e', 'l', 'l', 'o'};
```

c[0]~c[4]的值依次为H、e、l、l、o，其他的数组元素均为空字符。

如果提供的初值个数与预定的数组长度相同，在定义时可以省略数组长度，系统会自动根据初值的个数确定其长度。如:

```
char c[ ]={ 'I', ' ', 'a', 'm', ' ', 'a', ' ', 's', 't', 'u', 'd', 'e', 'n', 't', '.'};
```

数组c的长度自动定为15。用这种方法可以不必人工去数字符的个数，尤其在字符个数较多时，较为方便。

也可以定义和初始化一个二维字符数组，如:

```
char star[3][5]={{' ', ' ', '*'},{' ', '*', ' ', '*'},{'*', ' ', '*', ' ', '*'}};
```

### 7.2.3 字符数组的使用

下面将通过例子说明字符数组的使用。

**例7.19** 输出一个字符数组。

```
#include <iostream.h> // Li0719.cpp
void main( )
{   char c[15]={ 'I', ' ', 'a', 'm', ' ', 'a', ' ', 's', 't', 'u', 'd', 'e', 'n', 't', '.'};
    for(int i=0; i<15; i++) cout<<c[i];
    cout<<endl;
}
```

程序运行结果为:

```
I am a student.
```

**例7.20** 输出一个三角形图形。

```
#include <iostream.h>                    // Li0720.cpp
void main( )
{   char star[ ][5]={{' ', ' ', '*'},{' ', '*', ' ', '*'},{'*', ' ', '*', ' ', '*'}};
    for(int i=0; i<3; i++)
    {   for(int j=0; j<i+3; j++)         // 当输出空字符时，什么也不输出
                cout <<star[i][j];
            cout<<endl;
    }
}
```

程序运行结果为:

```
   *
  * *
 * * *
```

### 7.2.4 字符串和字符串结束标志

在C++语言中，用字符数组来处理字符串。例7.19就是用一个一维字符数组来存放字符串“I am a

student." 中的字符的。这个字符串的实际长度与字符个数相等。一般情况下，人们关心的是有效字符的长度而不是字符数组的长度。例如，定义一个字符数组长度为50，而实际有效字符只有20。为了测定字符串的实际长度，C++语言规定以字符'\0'作为 "字符串结束标志"。如果一个字符串，其中第10个字符为'\0'，则此字符串的有效字符为9个。也就是说，在遇到字符'\0'时，表示字符串结束，由它前面的字符组成字符串。

系统对字符串常量也自动加一个'\0'作为结束符。例如 "I am a student." 共15个字符，但在内存中占16个字节，最后一个字节'\0'是由系统自动加上的。'\0'的ASCII码为0，ASCII为0的字符是一个 "空操作符"，用它作为字符串结束标志不会产生附加的操作或增加有效的字符，而只起标志作用。字符串作为一维数组存放在内存中。

在程序中一般依靠检查'\0'的位置来判定字符串是否结束，而不是根据数组的长度来决定字符串长度。使用时，字符数组的长度应大于字符串的实际长度；如果一个字符数组用于存放多个字符串，则必须使得字符数组的长度大于最长的字符串的实际长度。

在了解了C++语言中处理字符串的方法以后，我们再补充一种字符数组初始化的方法，即用字符串常量来初始化字符数组。例如:

```
char c[ ]={"How are you?"};                                    (1)
```

也可以省略花括号，直接写成:

```
char c[ ]="How are you?";                                      (2)
```

这两种方法等价于:

```
char c[ ]={'H', 'o', 'w', ' ', 'a', 'r', 'e', ' ', 'y', 'o', 'u', '?', '\0' };    (3)
```

但(1)和(2)两种方法直观、方便，符合人们的习惯。注意数组c的长度不是12，而是13，因为在字符串常量的最后由系统加上了一个'\0'。

`char c[ ]={'H', 'o', 'w', ' ', 'a', 'r', 'e', ' ', 'y', 'o', 'u', '?'};` 与(1)、(2)、(3)不等价，前者的长度为13，后者的长度为12。如果有:

```
char c[10]={"Good!"};
```

数组c的前5个元素为'G'、'o'、'o'、'd'、'!'，第6个元素为'\0'，后4个元素也为空字符，如图7-8所示。

| G | o | o | d | ! | \0 | | | | |
|---|---|---|---|---|---|---|---|---|---|

图7-8 字符串的存储

需要说明的是: 字符数组并不要求它的最后一个字符为'\0'，甚至可以不包括'\0'。像以下这样写是完全合法的:

```
char c[5]={'G', 'o', 'o', 'd', '!'};
```

但只能将其处理成字符数组，而不能处理成字符串。

是否需要加'\0'，完全根据需要决定。但是由于系统对字符串常量自动加一个'\0'；因此，人们为了使处理方法一致，便于测定字符串的长度，以及在程序中进行相应的处理，在字符数组中也常常人为地加上一个'\0'。如:

```
char c[6]={'G','o','o','d','!','\0'};
```

### 7.2.5 字符数组的输入／输出

字符数组的输入／输出可以有两种方式，如下所示。

### 1. 逐个字符输入／输出

举例如下:

```
char s[10];
cout<<"输入10个字符:"<<endl;
for(int i=0; i<10; i++) cin>>s[i];
for(i=0; i<10; i++)cout<<s[i]<< '\t';
…
```

### 2. 将整个字符串输入或输出

对于一维字符数组的输入，在cin中只给出数组名，输出时在cout中也只给出数组名即可。

**例7.21**　将两个字符串分别输入到两个字符数组中，并将这两个数组中的字符串输出。

```
#include <iostream.h> //Li0721.cpp
void main( )
{   char str1[20], str2[20];
    cout<<"输入两个字符串: "<<endl;
    cin>>str1;
    cin>>str2;
    cout<<"字符串1="<<str1<<endl;
    cout<<"字符串2="<<str2<<endl;
}
```

程序运行情况如下:

```
输入两个字符串:
China Good<Enter>
字符串1=China
字符串2=Good
```

从输出结果可以看出，字符串“China”送给str1，“Good”送给str2。

注意:

(1) 输出字符不包括结束符'\0'。

(2) 用此法输出字符串时，输出项是字符数组名，而不是数组元素名。如果输出项是数组元素名，则输出的是数组元素。

(3) 如果数组长度大于字符串实际长度，输出时也只是遇到'\0'结束。如:

```
char c[10]={ "Good"};
cout<<c;
```

只输出“Good”4个字符，而不是输出10个字符。

(4) 如果一个字符数组中包含一个以上的'\0'，则遇到第一个'\0'时输出就结束。

(5) 在输入字符串时，遇到空格字符或换行符（Enter键），则认为一个字符串结束，接着的非空格字符作为下一个新的字符串开始。当要把输入的一行作为一个字符串送到字符数组中时，则要使用函数cin.getline( )。该函数的第一个参数为字符数组名，第二个参数减1为允许输入的最大字符个数。

**例7.22**　用函数cin.getline( )进行字符串的输入。

```
#include <iostream.h>                    //Li0722.cpp
void main( )
{   char str1[80];
    char str2[4]={'G', 'o', 'o', 'd'};
    cout<<"请输入一行字符串: "<<endl;
    cin.getline(str1, 80);               //表示最多输入79个字符，系统自动在最后加一个'\0'
    cout<<"str1="<<str1<<endl;
```

```
    cout<<"str2="<<str2<<endl;
}
```

程序运行情况如下:

```
请输入一行字符串:
I am a student. <Enter>
str1= I am a student.
str2=GoodI am a student.
```

从上例中可以看出:

(1) 通过cin.getline( )函数可以输入带空格的字符串。当输入行中的字符个数小于80时，将实际输入的字符串（不包括换行符）全部送给str1；当输入行中的字符个数大于或等于80时，只取前面的79个字符送给字符串。

(2) str2初始化是按逐个字符的方式进行的，而不是按字符串的方式初始化的，所以str2中没有字符串结束标志。把str2中的字符作为字符串输出时，由于最后一个字符'd'后没有字符串结束标志，仍把紧跟其后存储空间中的值作为字符输出（存储空间如图7-9所示），直至遇到字符串结束标志为止。所以当把字符数组中的字符作为字符串输出时，必须保证数组中包含字符串结束标志。

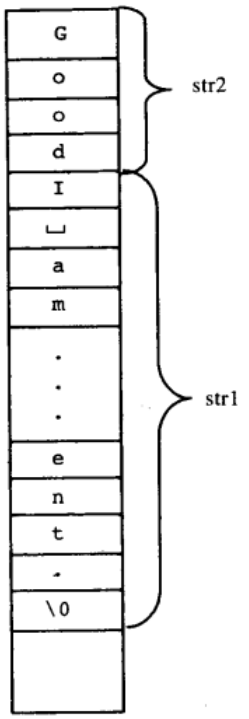



图7-9 存储空间分配情况

### 7.2.6 字符串处理函数

C++编译系统提供的字符串处理函数在string.h头文件中作了说明，要调用字符串处理函数时，需要包含string.h文件。下面介绍一些常用的字符串处理函数的功能及使用方法。

#### 1. 求字符串长度函数strlen(字符串)

strlen是英文STRing LENgth（字符串长度）的缩写，该函数的实参可以是字符数组名，也可以是字符串。其功能是求出一个字符串中包含的有效字符的个数（不包括字符'\0'）作为字符串的长度。例如:

```
char s1[80]= "China";
cout<<strlen(s1)<< '\n';            //输出结果为5
cout<<strlen("大学生")<<'\n';       //输出结果为6
```

**例7.23** 计算已知字符串的长度。

```
#include<iostream.h> //Li0723.cpp
#include<string.h>
void main( )
{   char s1[ ]="How do you do!";
    char s2[ ]="Hello!";
    char s3[80];
    cout<<"Input a word:";
    cin>>s3;
    cout<<"s1:"<<strlen(s1)<<endl;
    cout<<"s2:"<<strlen(s2)<<endl;
    cout<<"s3:"<<strlen(s3)<<endl;
}
```

程序运行情况为:

```
Input a word:good<Enter>
s1:14
```

```
s2:6
s3:4
```

### 2. 字符串拷贝函数strcpy(字符数组1, 字符串2)

strcpy是英文STRing CoPY（字符串拷贝）的缩写，该函数的功能是将字符串2拷贝到字符数组1中去。例如:

```
char str1[80]={ "I am a student."};
char str2[80];
strcpy(str2, str1);
```

将str1中的字符串拷贝到str2中，使str2包含字符串“I am a student.”。

说明:

（1）字符数组1的长度必须大于等于字符串2的长度。

（2）拷贝时连同字符串后面的'\0'一起拷贝到字符数组1中。

（3）不能用赋值语句将一个字符串常量或字符数组直接赋给一个字符数组，如下面两行是不合法的:

```
char str1[80], str2[80];
str1={"Good"};
str1=str2;
```

而只能用strcpy函数处理。用一个赋值语句只能将一个字符赋给一个字符型变量或字符型数组元素，如下面是合法的:

```
char c[5], c1, c2;
c1='A'; c2='B';
c[0]='G'; c[1]= 'o'; c[2]= 'o'; c[3]= 'd'; c[4]= '\0';
```

### 3. 字符串连接函数strcat(字符数组名1, 字符数组名2)

strcat是英文STRing CATenate（字符串连接）的缩写，该函数的功能是把字符数组2中的字符串2连接到字符数组1中字符串1的后面，结果放在字符数组1中，对字符数组2中的内容没有影响。例如:

```
char s1[20]= "one";
char s2[20]= "two";
char s3[20]= "three";
strcat(s1, s2);
strcat(s1, s3);
```

s1中的字符串为"onetwothree"。

该函数中的第二个参数也可以是一个字符串。

### 4. 字符串比较函数strcmp(字符串1, 字符串2)

strcmp是英文STRing CoMPare（字符串比较）的缩写，该函数的两个实参可以是字符数组名，也可以是字符串。其功能是用来比较两个字符串是否相等。从两个字符串的第一个字符开始自左至右逐个字符进行比较，这种比较是按字符的ASCII码值的大小进行的，直到出现两个不同的字符或遇到字符串的结束标志'\0'为止。如果两个字符串中的字符均相同，则两个字符串相等，函数返回值为0。当两个字符串不同时，则以自左至右出现的第一个不同字符的比较结果作为两个字符串的比较结果，并将该比较结果作为函数的返回值返回。如果第一个字符串大于第二个字符串，则返回值为1。如果第一个字符串小于第二个字符串，则返回值为－1。例如:

```
strcmp("Student", "Student");
strcmp("student", "Student");
strcmp("Student", "student");
```

第一个函数调用比较结果为0，第二个函数调用比较结果为1，第三个函数调用比较结果为－1。

### 5. 大写字母变成小写字母函数strlwr(字符数组名)

strlwr是英文STRing LoWeRcase（字符串小写）的缩写，该函数将字符数组中存放的字符串中的所有大写字母变成小写字母。例如:

```
char s1[ ]="Student";
strlwr(s1);
```

s1数组中的字符串全部变成小写字母。

### 6. 小写字母变成大写字母函数strupr(字符数组名)

strupr是英文STRing UPpeRcase（字符串大写）的缩写，该函数将字符数组中存放的字符串中的所有小写字母变成大写字母。例如:

```
char s1[ ]="Student";
strupr (s1);
```

s1数组中的字符串全部变成大写字母。

### 7. 函数strncpy(字符数组名1, 字符串2, len)

该函数的功能是将字符串2的前len个字符复制到字符数组1的前len个字符空间中。其中第二个参数可以是数组名，也可以是字符串，第三个参数为正整数。当字符串2的长度小于len时，把字符串2全部拷贝到第1个参数所指定的数组中。例如:

```
char s1[80], s2[80];
strncpy(s1, "student", 4);
strncpy(s2, "teacher", 10);
```

第1个strncpy函数仅拷贝前4个字符，s1中为stud。第2个strncpy函数由于字符串"teacher"的长度小于10，则将该字符串全部拷贝到s2中。

### 8. 函数strncmp(字符数组名1, 字符串2, len)

该函数的功能是比较两个字符串中前len个字符。其中前两个参数均可为数组名或字符串，第3个参数为正整数。若字符串1或字符串2的长度小于len时，该函数的功能与strcmp( )相同。当两个字符串的长度均大于len时，len为最多要比较的字符个数。例如:

```
cout<<strncmp("English", "England", 4)<<endl;
```

因为两个字符串的前4个字符相同，所以输出的值为0。

## 7.2.7 字符数组应用举例

**例7.24** 编写一个函数my_strcpy，完成与系统标准库函数strcpy( )相同的功能。

```
#include <iostream.h> // Li0724.cpp
void main( )
{   void my_strcpy(char [ ], char [ ]);
    char s1[80], s2[80];
    cout<<"请输入一个字符串:"<<endl;
    cin.getline(s2, 80);
    my_strcpy(s1, s2);
    cout<<"拷贝后的两个字符串分别为:"<<endl;
    cout<<s1<<endl;
    cout<<s2<<endl;
}
void my_strcpy(char s1[ ], char s2[ ])
{   int i=0;
    while(s2[i]!='\0')
```

```
        sl[i]=s2[i++];
    sl[i]='\0';
}
```

该程序比较简单，请读者自己分析。

**例7.25** 输入一行字符，统计其中的单词个数，单词之间用空格隔开。

求单词数的方法，顺序扫描数组元素，若当前字符是非空格，而其前一个字符是空格，则单词数加1。

```
#include <iostream.h>                              //Li0725.cpp
#include <string.h>
int numwords(char string[ ])
{     int i, j, num=0;
      j=strlen(string);
      for (i=0; i<j; )
      {       while(string[i]==' ')i++;            //滤掉多个连续的空格
              if (i<j) num++;                      //单词数加1
              while(string[i]!=' '&&i<j) i++;      //跳过一个单词
      }
return num;
}
void main( )
{  char string[80];
   int num;
   cout<<"输入一行字符:";
   cin.getline(string, 80);
   num=numwords(string);
   cout<<"输入的字符串为:"<<string<<endl;
   cout<<"字符串中包含的单词个数为:"<<num<<endl;
}
```

运行情况如下:

```
输入一行字符:I am a student. <Enter>
输入的字符串为:I am a student.
字符串中包含的单词个数为:4
```

**例7.26** 有三个字符串，要求找出其中最小者。

```
#include <iostream.h>      //Li0726.cpp
#include <string.h>
void main( )
{  char string[80];
   char str[3][80];
   cout<<"请输入三个字符串: \n";
   for (int i=0; i<3; i++)
          cin.getline(str[i], 80);
   if (strcmp(str[0], str[1])<0) strcpy(string, str[0]);
   else strcpy(string, str[1]);
   if (strcmp(str[2], string)<0) strcpy(string, str[2]);
   cout<<"最小的字符串为: "<<string<<endl;
}
```

运行情况如下:

```
请输入三个字符串:
China<Enter>
```

```
American<Enter>
Japan<Enter>
最小的字符串为: American
```

**例7.27** 编写一个程序，计算一个字符串中子串出现的次数。

```
#include <iostream.h>                          // Li0727.cpp
#include <string.h>
int count(char mstr[ ], char sstr[ ])
{   int i, j, k, num=0;                        // num用于统计子串在主串中出现的次数
    for(i=0; mstr[i]!='\0'; i++)               // 从主串开头位置开始扫描
    {      for(j=i, k=0; mstr[j]!='\0'&&sstr[k]==mstr[j]; j++, k++);
                                               // 空循环体，比较与子串是否相同
        if(sstr[k]=='\0') num++;
    }
    return num;
}
void main( )
{   int num;
    char mstring[80], sstring[80];
    cout<<"请输入主串:";
    cin.getline(mstring, 80);
    cout<<"请输入子串:";
    cin.getline(sstring, 80);
    num=count(mstring, sstring);
    if(num>0)
           cout<<"子串在主串中出现的次数为:"<<num<<endl;
    else
           cout<<"子串不在主串中"<<endl;
}
```

运行情况如下:

```
请输入主串:abcdddabcfff<Enter>
请输入子串:abc<Enter>
子串在主串中出现的次数为:2
```

## 习题7

1. 有两个一维数组中的元素已按升序排列，编写一个函数将两个数组中的元素归并成一个数组，其中的元素仍然按升序排列。
2. 假定在一个整型数组中，每一个元素都是不超过两位的正整数。试编程统计该数组全部元素中数字0，1，…，9各出现多少次。
3. 编写一个函数判断一个整数是否为回文数。如果一个数从正的方向读和从反的方向读的结果相同，则该数就是回文数。例如，整数4，66，676，1234321都是回文数。
4. 编写一个函数，判断给定的字母是否是元音。编写主函数输入一系列字符，统计其中元音字母的个数。
5. 编写一个函数实现数制转换。在主函数中输入一个十进制数，输出相应的十六进制数。要求用数组实现。
6. 输入若干个数到一维数组中，按升序排序后输出。
7. 编写一个程序，输入一个正整数，输出其相反数，如1234的相反数是4321。用数组实现。
8. 寻找二维数组中的鞍点，即该位置上的元素是所在行上最大的元素，是所在列上最小的元素。也可能没有鞍点。

9. 编写一个函数求二维数组中周边元素之和。
10. 编写一个函数my_strlwr( )，将一个字符串中的字符全部转换成小写字母。
11. 编写字符串反转函数。该函数的功能是将指定字符串中的字符顺序颠倒排列，然后再编写主函数进行验证。
12. 编写一个程序，判定一个字符串是否是另一个字符串的子串，若是，则返回子串在主串中的位置。
13. 一篇文章有若干行，以空行作为输入结束的条件。统计一篇文章中的单词the（不管大小写，单词the是由空格隔开的）的个数。

# 第8章 结构体、共用体和枚举类型

迄今为止，已经介绍了基本类型的变量（如整型、实型、字符型变量等），也介绍了一种导出数据类型——数组，数组中的各个元素属于同一数据类型。

但是只有这些数据类型是不够的。有时需要将描述一个对象的相关属性组成一个整体，以便引用。这些组合在一个整体中的数据是互相关联的。例如，一个学生的学号、姓名、性别、年龄、成绩等，这些项都与某一学生相联系，如图8-1所示。可以看到性别（sex）、年龄（age）、成绩（score）是属于学号为“23901”和姓名为“LiMing”的学生的。如果将num、name、sex、age、score分别定义为互相独立的简单变量，难以反映它们之间的内在联系。应当把它们组织成一个组合项，在一个组合项中包含若干个类型不同（也可以相同）的数据成员。C++语言允许用户自己构造这样一种新的数据类型，这种数据类型称为结构体（structure）。

| num | name | sex | age | score |
|---|---|---|---|---|
| 23901 | LiMing | M | 19 | 85 |

图8-1 学生信息

除了结构体外，C++语言还提供了共用体、枚举类型等导出类型。本章介绍结构体、共用体和枚举类型的定义方法和应用。

## 8.1 结构体的定义及应用

### 8.1.1 结构体类型的定义

假设程序中要用图8-1所表示的数据结构，但是C++中没有提供这种现成的数据类型，因此用户必须在程序中建立所需的结构体类型。

定义一个结构体类型的一般形式为:

```
struct <结构体名>
{
        <成员列表>
};
```

其中，结构体名由标识符组成，大括号内是该结构体中的各个成员，如上例中的num、name、sex等都是成员。对各个成员都应进行类型说明，即:

```
<类型名> <成员名>;
```

成员名的命名规则与变量名相同。

例如:

```
struct student
{   int num;
    char name[20];
    char sex;
    int age;
    float score;
};
```

注意，不要忽略最后的分号。上面定义了一个新的结构体类型struct student（struct是声明结构体类

型时所必须使用的关键字，不能省略），它向编译系统声明这是一个“结构体类型”，它包含num、name、sex、age、score等不同类型的数据成员。struct student（或student）是一个用户定义的新的数据类型名，它和系统提供的标准数据类型具有同样的地位和作用。

### 8.1.2 结构体类型变量的定义

前面定义了一个结构体类型，它是一个抽象的类型，其中并没有具体的数据，系统对它也不分配实际内存单元。为了能在程序中使用结构体类型的数据，应当定义结构体类型的变量，并在其中存放具体的数据。定义变量的格式为:

<类型名> <变量名列表>

类型名可以是标准的或导出的。对结构体类型变量，可以采用以下三种方法定义。

#### 1. 先定义结构体类型再定义变量

如前面已定义了一个结构体类型struct student，则可以用它来定义变量。如:

```
struct student stu1, stu2; 或student stu1, stut2;
```

其中struct student（或student）为结构体类型名，stu1和stu2为结构体变量名。定义了stu1和stu2为struct student类型的变量，即它们具有struct student类型的结构，如图8-2所示。

| | | | | | |
|---|---|---|---|---|---|
| stu1 | 23902 | Zhang li | F | 19 | 85 |
| stu2 | 23924 | Li lie | M | 20 | 88 |

图8-2　struct student类型变量的存储示意

在定义了结构体变量后，系统会为它分配内存单元。

#### 2. 在定义类型的同时定义变量

其一般形式为:

```
struct <结构体名>
{
    <成员列表>
}<变量名列表>;
```

例如:

```
struct student
{   int num;
    char name[20];
    char sex;
    int age;
    float score;
}stu1, stu2;
```

它的作用与第一种方法相同，即定义了两个struct student类型的变量stu1、stu2。

#### 3. 直接定义结构体类型变量

其一般形式为:

```
struct
{
    <成员列表>
}<变量名列表>;
```

即不出现结构体名。

例如:

```
struct
{   int num;
    char name[20];
    char sex;
    int age;
    float score;
}stu1, stu2;
```

由于前两种方法都定义了结构体的类型名，因此在程序中可以用该类型名来定义其他结构体类型变量；而第三种方法没有定义类型名，则无法再定义这种类型的变量。建议使用第一种方法来定义结构体类型的变量。如果程序规模较大，往往将对结构体的定义集中放到一个头文件中。如果源程序需要用到此结构体类型，则可用#include命令将该头文件包含到本文件中。这样便于装配、修改及使用。

另外，在定义结构体类型变量的同时，可以指定其存储类型。例如:

```
static student stud3;
auto student stud4;
extern student stud5;
```

以上定义的结构体变量都是合法的。

在定义结构体类型变量的同时，可以对结构体类型的变量进行初始化。其初始化的方法是用花括号将每一个成员的值括起来。例如:

```
student stud1={23901, "李力", 'M', 19,86.5};
```

表示stud1的成员num初始化为23901，成员name初始化为"李力"，成员sex初始化为'M'，成员age初始化为19，成员score初始化为86.5。注意，初始化时，在花括号中列出的值的类型及顺序必须与该结构体类型定义中所说明的结构体成员一一对应。例如:

```
student stud1={"23901", "李力", 'M', 19, 86};
```

则编译出错，因stud1的成员num是整型数，而这里为字符串。

### 8.1.3 结构体类型变量及其成员的引用

在定义了结构体变量后，可以引用这个变量，有两种引用方式，如下所示。

#### 1. 引用成员

引用结构体变量中的成员的方法为:

```
<结构体变量名>. <成员名>
```

例如，stud1.num表示stud1变量中的num成员，可以对结构体变量的成员赋值。例如:

```
stud1.num=23901;
```

“.”是成员运算符，把stud1.num作为一个整体来看待。上面的赋值语句的作用是将整数23901赋给stud1变量中的成员num。

结构体变量的成员可以像普通变量一样进行各种运算（根据其类型决定可以进行的运算）。例如:

```
stud1.num++;
stud1.sex=stud2.sex;
```

#### 2. 引用整体

同类型的结构体变量之间可以直接赋值。这种赋值等同于各个成员的依次赋值。如定义以下的结构体类型:

```
struct t
{   int i, j;
```

```
    char name[10];
};
t t1={12, 48, "LiLi"}, t2;
t2=t1;
```

其中“t2=t1;”等同于:

```
t2.i=t1.i;
t2.j=t1.j;
strcpy(t2.name, t1.name);
```

在引用结构体变量及其成员时，应注意以下几点:

(1) 不能将结构体变量作为一个整体进行输入／输出。例如，已定义stud1和stud2为结构体变量并且它们已有值，则不能这样引用:

```
cout<<stud1;
```

只能对结构体变量中的各个成员分别进行输入／输出。例如:

```
cout<<stud1.num;
```

注意: 在学习了面向对象部分的运算符重载后，如果对该结构体类型进行了运算符<<和>>的重载，这时就可以将结构体变量整体输入／输出了。

(2) 如果成员本身又是一个结构体类型，则要用若干个成员运算符，逐级地找到最低一级的成员。只能对最低一级的成员进行赋值或存取以及运算。例如，对前面定义的结构体变量stud1，可以这样访问各成员:

```
stud1.num
stud1.birthday.day
```

注意，不要用stud1.birthday来访问stud1变量中的成员birthday，因为birthday是一个结构体变量。

(3) 结构体变量可以作为函数的参数，也可以作为函数的返回值。当函数的形参与实参为结构体变量时，这种结合方式属于值调用方式，即属于值传递，即形参的变化不影响实参。如果用结构体变量的引用做参数，则直接对实参的结构体变量进行修改。

**例8.1** 有一个结构体变量stu，内含学生学号、姓名和四门课的成绩。要求在main函数中赋值，在另一函数print中将它们输出。这里用结构体变量做函数参数。

```
#include <iostream.h>      //Li0801.cpp
#include <string.h>
struct student
{   int num;
    char name[20];
    float score[4];
};
void main( )
{   void print(student);
    student stud;
    stud.num=2468;
    strcpy(stud.name, "LiWen");
    stud.score[0]=68.5;
    stud.score[1]=90;
    stud.score[2]=78.5;
    stud.score[3]=85.5;
    print(stud);
}
```

```
void print(student stud)
{   cout<<"学号\t\t"<<stud.num<<'\n'<<"姓名\t\t"<<stud.name<<'\n';
    cout<<"数学成绩\t"<<stud.score[0]<<'\n'<<"英语成绩\t"<<stud.score[1]<<'\n';
    cout<<"程序设计成绩\t"<<stud.score[2]<<'\n'<<"物理成绩\t"<<stud.score[3]<<endl;
}
```

运行结果为:

```
学号            2468
姓名            LiWen
数学成绩        68.5
英语成绩        90
程序设计成绩    78.5
物理成绩        85.5
```

struct student被定义为外部类型，同一源文件中的各个函数都可以用它来定义变量的类型。main函数中的stud是struct student类型变量，print函数中的形参stud也是struct student类型变量。在main函数中对stud的各成员赋值。在调用print函数时以stud为实参向形参stud实行“值传递”。在print函数中输出结构体变量stud各成员的值。

关于结构体类型，有几点说明:

(1) 类型与变量是不同的概念。类型是抽象的，而变量是具体的。在编译时，对类型是不分配空间的，只对变量分配空间。使用时只能对变量赋值、存取或运算，而不能对一个类型赋值、存取或运算。

(2) 对结构体中的成员，可以单独使用，它的作用与地位相当于普通变量。

(3) 成员也可以是一个结构体变量。

如:

```
struct date
{   int year;
    int month;
    int day;
};
struct student
{   int num;
    char name[20];
    char sex;
    struct date birthday;            //birthday是struct date类型
    char addr[40];
} stu1, stu2;
```

先定义一个struct date类型，它代表“日期”，包括三个成员: year (年)、month (月)、day (日)。然后在定义struct student类型时将成员birthday指定为struct date类型。struct student的结构如图8-3所示。已定义的类型struct date可以被用来定义其他类型的成员。

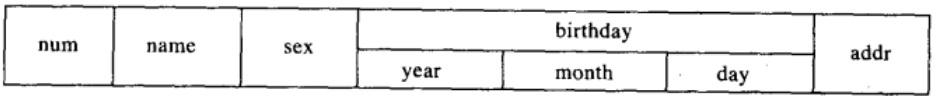



图8-3　struct student的结构

(4) 结构体类型及其变量的使用与标准数据类型及其变量的使用方式是一样的，包括:

① 变量的作用域和存储类别。

② 可以作为函数的参数及返回值。

③ 可以定义结构体数组等。

### 8.1.4 结构体数组

一个结构体变量中可以存放一组数据（如一个学生的学号、姓名、成绩等）。如果有10个学生的数据需要参加运算，则应该用数组，这就是结构体数组。结构体数组与前面介绍过的标准数据类型数组的不同之处在于，每个数组元素都是一个结构体类型的变量，它们都分别包括各个成员项。

定义结构体数组的方法与定义结构体变量的方法类似，也可以有三种方法，只要在每一种方法的基础上，增加数组维数的说明即可。如:

```
struct student
{   int num;
    char name[20];
    char sex;
    int age;
    float score;
    char addr[30];
};
student stud[4];
```

以上定义了一个数组stud，其元素为struct student类型数据，数组有四个元素。也可以直接定义一个结构体数组，如:

```
struct student
{   int num;
    char name[20];
    char sex;
    int age;
    float score;
    char addr[30];
} stud[4];
```

或

```
struct
{   int num;
    char name[20];
    char sex;
    int age;
    float score;
    char addr[30];
} stud[4];
```

在定义结构体数组时可以对结构体数组进行初始化，方法与数组的初始化方法雷同：第一种方法是将每个元素的成员值用花括号括起来，再将数组的全部元素值用一对花括号括起来，第二种方法是在一个花括号内依次列出各个元素的成员值。如:

```
struct student
{    int num;
     char name[20];
     char sex;
     int age;
     float score;
     char addr[30];
} stud[4] =
{ {23901, "Zang Li", 'F', 19, 78.5, "35 Shanghai Road"},
{23902, "Wang Fang", 'F', 19, 92, "101 Taiping Road"},
```

```
{23905, "Zhao Qiang", 'M', 20, 87, "56 Ninghai Road"},
{23908, "Li Hai", 'M', 19, 95, "48 Jiankang Road"}};
```

或

```
struct student
{    int num;
     char name[20];
     char sex;
     int age;
     float score;
     char addr[30];
} stud[4]={23901, "Zang Li", 'F', 19, 78.5, "35 Shanghai Road",
23902, "Wang Fang", 'F', 19, 92, "101 Taiping Road",
23905, "Zhao Qiang", 'M', 20, 87, "56 Ninghai Road",
23908, "Li Hai", 'M', 19, 95, "48 Jiankang Road"};
```

结构体数组初始化后，在内存中存放的逻辑示意图如图8-4所示。

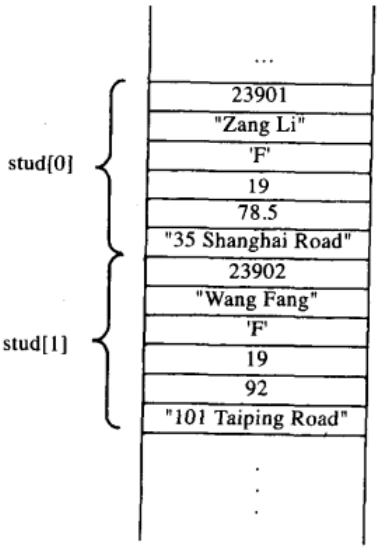



图8-4 结构体在数组内存中的存放情况

下面举一个简单的例子来说明结构体数组的定义和引用。

**例8.2** 建立一个学生档案的结构体数组，描述一个学生的信息：学号、姓名、成绩，并输出已建立的学生档案。

```
#include <iostream.h>      //Li0802.cpp
struct student
{
    int num;
    char name[20];
    float score;
};
student Input( )
{   student stud;
    cout<<"请输入学号、姓名和成绩:";
    cin>>stud.num>>stud.name>>stud.score;
    return stud;
}
void Output(student stud)
{  cout<<"学号为  "<<stud.num<<'\n'<<"姓名为  "<<stud.name<<'\n'
       <<"成绩为  "<<stud.score<<endl;
}
void main( )
{  student studs[3];
   for (int i=0; i<3; i++)
          studs[i]=Input( );
   for(i=0; i<3; i++)
         Output(studs[i]);
   cout<<endl;
}
```

在上例中，函数Input( )返回输入值的结构体变量，增加将返回值赋给数组元素的时间开销。若使用引用类型的结构体变量做形参，程序的运行效率会高一些，即系统的开销要小一些，因为不涉及结构体元素整体赋值的时间开销。改为引用调用方式后，程序如下：

```
#include <iostream.h>
struct student
{    int num;
     char name[20];
     float score;
};
void Input(student &stud)
{    cout<<"请输入学号、姓名和成绩:";
     cin>>stud.num>>stud.name>>stud.score;
}
void Output(student &stud)
{    cout<<"学号为  "<<stud.num<<'\n'<<"姓名为  "<<stud.name<<'\n'
         <<"成绩为  "<<stud.score<<endl;
}
void main( )
{    student studs[3];
     for (int i=0; i<3; i++)
           Input(studs[i]);
     for(i=0; i<3; i++)
           Output(studs[i]);
     cout<<endl;
}
```

该程序与前一个程序相比，不但效率提高了，而且程序也紧凑了。引用类型将在第9章详细介绍。

**例8.3** 求若干学生的平均成绩。

```
#include <iostream.h>      //Li0803.cpp 结构体数组做函数参数
struct  stud
{    int    num;
     char   name[20];
     int    age;
     char   sex;
     int    score;
};
float average(stud studs[], int n)
{
     float aver=0;
     for(int i=0; i<n; i++)
       aver += studs[i].score;
     aver /= n;
     return aver;
}
void main(void)
{    stud studs[4]={ {020110101, "Wu", 19, 'M', 80}, {020110102, "Li", 18, 'F', 95},
                     {020110103, "Zhang", 18, 'F', 78}, {020110104, "Zhao", 20, 'M', 88} };
     float  aver ;
     aver=average(studs, 4);
     cout << "平均成绩为: "<< aver << endl;
}
```

结构体数组做函数参数与简单变量数组做函数参数类似，请读者体会它的用法。

## 8.2 共用体的定义及应用

### 8.2.1 共用体类型及其变量的定义

有时需要使几个不同类型的变量共用同一段存储单元。例如，可把一个短整型变量i、一个字符型变量c、一个实型变量f放在同一个地址开始的内存单元中（见图8-5）。以上三个变量在内存中占的字节数不同，但都从同一地址开始（图中假设地址为2000）存放。也就是使用覆盖技术，几个变量互相覆盖，在某一时刻，只有一个变量有用。这种使几个不同的变量共占同一段内存的结构称为“共用体”类型的结构，“共用体”有时也被称为“共同体”。定义共用体类型变量的一般形式为:

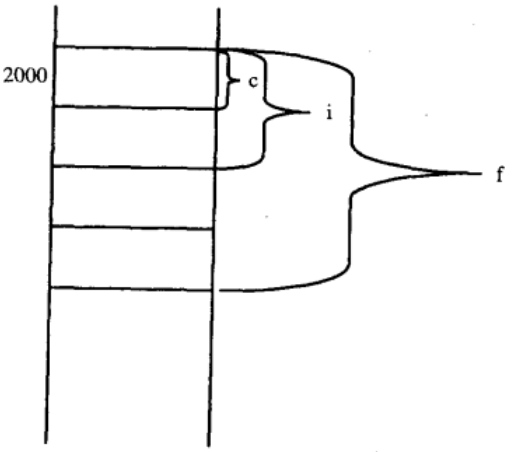



图8-5 共用体存放内存示意图

```
union <共用体名>
{
    <成员列表>
}<变量列表>;
```

例如:

```
union data
{   short int i;
    char c;
    float f;
} a, b, c;
```

也可以先定义一个union data类型，再将a、b、c定义为union data类型的变量。当然也可以直接定义共用体变量，如:

```
union
{ short int i;
  char c;
  float f;
} a, b, c;
```

可以看到，“共用体”与“结构体”的定义形式类似，但它们的含义不同。结构体变量的每个成员分别占有其自己的内存单元。共用体变量所占的内存长度是所有成员中最长的成员的长度。例如，上面定义的“共用体”变量a、b、c共同占有这四个字节（因为一个整型变量和一个实型变量均占四个字节）。

### 8.2.2 共用体类型变量的引用

共用体类型也是一种新的数据类型，在做函数参数、定义共用体变量数组等情况下，与基本数据类型的使用方式相同。首先应定义共用体变量然后才能引用它，共用体类型变量的引用也有两种方式:

#### 1. 引用成员

例如，前面定义了共用体变量a、b、c，下面的引用方式是正确的:

```
a.i     // 引用共用体变量中的整型变量i
a.c     // 引用共用体变量中的字符变量c
a.f     // 引用共用体变量中的实型变量f
```

#### 2. 引用整体

同类型的共用体变量之间可以直接赋值。例如: 前面已定义了共用体变量a、b。

```
a.i=2;
b=a;     // 引用整体
```

但输入／输出时不能只引用共用体变量，例如:

```
cout<<a<<endl;
```

是错误的，a的存储区有好几种类型，分别占不同长度的存储区，仅仅用共用体变量名a，难以使系统确定究竟输出的是哪个成员的值。应该写成cout<<a.i或cout<<a.c等。

### 8.2.3 共用体数据类型的特点

共用体类型变量与结构体类型变量雷同，它不能直接进行输入／输出；用做函数的参数时，也是作为值传递，同类型的共用体变量之间可以相互赋值等。在使用共用体数据时应注意以下一些特点:

(1) 同一个内存段可以用来存放几种不同类型的成员，但在每一瞬时只能存放其中一种，而不是同时存放几种。也就是说，每一瞬时只有一个成员起作用，其他的成员不起作用。

(2) 共用体变量中起作用的成员是最后一次存放的成员，在存入一个新的成员后原有的成员就失去作用。例如有以下赋值语句:

```
a.i=10;
a.c='A';
a.f=1.8;
```

在完成以上三个赋值运算以后，只有a.f是有效的。a.i和a.c失去意义。注: "cout<<a.i;" 可以运行，但是将a.f最低的两个字节解释为整数输出。因此在引用共用体变量时应十分注意当前存放在共用体变量中的究竟是哪个成员。

(3) 共用体变量的起始地址和它的各成员的起始地址都是同一地址。

(4) 如果在定义共用体变量时对它初始化，则只允许有一个数赋给第一个成员。例如:

```
union
{   int i;
    char c;
    float f;
} a = {10, 'A', 1.8};              // 不能这样初始化

union
{   int i;
    char c;
    float f;
} a={10};                          // 能这样初始化
```

(5) 共用体类型可以出现在结构体类型定义中，也可以定义共用体数组。反之，结构体也可以出现在共用体类型定义中，数组也可以作为共用体的成员。

定义共用体的目的有两个: ①节省空间; ②特殊效果，比如分别取出一个整数的四个字节。

**例8.4** 分别取出一个整数的四个字节。

```
#include <iostream.h>                          // Li0804.cpp
void main( )
{  union
   {    int i;
        char c[4];
   }a;
   cout<<"请输入一个整数:";
   cin>>a.i;
   cout<<"整数的四个字节中的数分别为:";
   for(int k=3; k>=0; k--)
          cout<<(int)a.c[k]<<'\t';         // 一个整数的四个字节分别对应字符数组的每个元素
```

```
    cout<<endl;
}
```

运行情况如下:

```
请输入一个整数:511<Enter>
整数的四个字节中的数分别为:0    0    1    -1
```

请读者自己分析为什么会出现负数?

## 8.3 枚举类型

为了对变量的取值范围做限制引入枚举类型，枚举类型也是一种构造数据类型。“枚举”就是将变量允许取值的范围一一列举出来，枚举变量的取值限于列举出来的值的范围内。

### 8.3.1 枚举类型的定义

定义枚举类型的一般形式为:

```
enum  <枚举类型名> {<枚举常量列表>};
```

其中，enum是一个关键字，枚举类型名的命名规则与一般标识符相同，枚举常量表由若干个枚举常量组成，多个枚举常量之间用逗号隔开。每个枚举常量是一个用标识符表示的整型或字符型常量。例如:

```
enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

其中，day是一个枚举类型名，该枚举常量列表中有七个枚举常量。每个枚举常量所表示的整型数值在默认的情况下，第一个为0，第二个为1，后一个总是前一个的值加1。枚举常量的值可以在定义时被显式赋值，被显式赋值的枚举常量将获得该值，没有被赋值的枚举常量按照后一个总是前一个的值加1的规则分别获得值。例如:

```
enum day {Sun=7, Mon=1, Tue, Wed, Thu, Fri, Sat};
```

这里Sun的值为7，Mon的值为1，Tue的值为2，Wed的值为3……Sat的值为6。

### 8.3.2 枚举类型变量的定义

定义一个枚举变量前，必须先定义一个枚举类型，枚举变量的定义形式如下:

```
enum <枚举类型名> <枚举变量名列表>;
```

其中，枚举变量名列表中，如有多个枚举变量名，则用逗号分隔。例如:

```
enum day day1, day2, day3;
```

这里，day是前面定义的枚举类型名，day1、day2和day3是三个枚举变量名，这三个枚举变量是属于枚举类型名为day的枚举变量，它们的值便是上面枚举常量列表中规定的七个枚举常量之一。

枚举变量的定义也可以与枚举类型的定义连在一起来写。上例可以写成:

```
enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat} day1, day2, day3;
```

### 8.3.3 枚举类型变量的使用

下面通过例子来说明枚举类型变量的使用。

**例8.5** 定义一个枚举类型triangle，其中的枚举常量有scalene（不等边三角形），isosceles（等腰三角形），equilateral（等边三角形），notriangle（非三角形）。编写程序，根据输入的三角形各边的长度，输出三角形的形状。

```
#include <iostream.h>                    //Li0805.cpp
void main( )
{   enum triangle{scalene, isosceles, equilateral, notriangle};
    triangle tri;
```

```
    int a, b, c;
    cout<<"请输入三角形的三条边:";
    cin>>a>>b>>c;
    if(a+b<=c||a+c<=b||b+c<=a) tri=notriangle;
    else if(a==b&&b==c) tri=equilateral;
    else if(a==b||b==c||a==c)tri=isosceles;
    else tri=scalene;
    switch(tri)
    {       case scalene : cout<<"为不等边三角形"<<endl; break;
            case isosceles : cout<<"为等腰三角形"<<endl; break;
            case equilateral : cout<<"为等边三角形"<<endl; break;
            case notriangle : cout<<"不能构成三角形"<<endl; break;
    }
}
```

运行情况如下:

```
请输入三角形的三条边:1 2 3<Enter>
不能构成三角形
请输入三角形的三条边: 3 4 5<Enter>
为不等边三角形
请输入三角形的三条边: 3 3 4<Enter>
为等腰三角形
请输入三角形的三条边: 3 3 3<Enter>
为等边三角形
```

**例8.6**　口袋中有红、黄、蓝、白、黑五种颜色的球若干个。每次从口袋中取出三个球，问得到三种不同的球的可能取法，打印出每种组合的三种颜色。

**解题思路**：因为球只能是五种颜色之一，而且要判断各球是否同色，所以用枚举类型变量来处理。

设取出的球为i、j、k。根据题意，i、j、k分别为五种颜色球之一，并要求i≠j≠k。可以用穷举法，逐个检查每一种可能的组合，从中找出符合要求的组合并输出。

```
#include <iostream.h>                                    //Li0806.cpp 取球
#include <iomanip.h>
enum color {red,yellow,blue,white,black};
void print(color col)
{   switch (col)
    {       case red: cout<<setw(10)<<"red";break;       //case后面可以是枚举常量
            case yellow: cout<<setw(10)<<"yellow";break;
            case blue: cout<<setw(10)<<"blue";break;
            case white: cout<<setw(10)<<"white";break;
            case black: cout<<setw(10)<<"black";break;
            default:break;
    }
}
void main( )
{   enum color pri;
    int i,j,k,n,loop;
    n=0;
    for(i=red;i<=black;i++)
          for(j=red;j<=black;j++)
                if(i!=j)
```

```
                for (k=red;k<=black;k++)
                    if((k!=i)&&(k!=j))
                    {   n=n+1;
                        cout<<setw(4)<<n;
                        for(loop=1;loop<=3;loop++)
                        {       switch(loop)
                                {       case 1 : pri=(enum color)i;break;
                                        case 2 : pri=(enum color)j;break;
                                        case 3 : pri=(enum color)k;break;
                                        default:break;
                                }
                        print(pri);
                        }
                        cout<<endl;
                    }
                cout<<setw(5)<<n;
}
```

运行结果为:

```
   1        red    yellow      blue
   2        red    yellow     white
   3        red    yellow     black
   4        red      blue    yellow
   5        red      blue     white
   6        red      blue     black
   7        red     white    yellow
   8        red     white      blue
   9        red     white     black
  10        red     black    yellow
  ......
  56      black      blue    yellow
  57      black      blue     white
  58      black     white       red
  59      black     white    yellow
  60      black     white      blue
  可能的组合数为 : 60
```

在实际使用中可以用整型常量代替枚举常量，但是显然没有枚举常量更直观，因为枚举常量都选用了“见名思意”的标识符，而且枚举变量的值限制在定义时规定的几个枚举常量范围内，如果赋予它一个其他的值，就会出现出错信息，便于检查。

## 习题8

1. 编写读取学生的学号、姓名和考试分数的程序。该程序输出每个学生的学号、姓名和相应的考试成绩，同时能查找并输出最高分的学生的学号、姓名和考试分数。
2. 定义一个结构体变量（包括年、月、日）。编写一个程序，该程序读入“年-月-日”形式的日期，并计算该日期是该年中的第几天。注意闰年问题。
3. 编写一个函数output，输出一个学生各门课的成绩。设计一个程序输出本班同学的成绩，假定学生信息有学号、姓名及五门课的成绩。

4. 为全班同学建立一个通讯录（用结构体数组实现），包括学号、姓名、家庭住址、电话号码、手机号码及E-mail地址。完成数据的输入和输出。
5. 定义描述三维坐标点(x, y, z)的结构体类型变量，完成坐标点的输入和输出，并求出两点之间的距离。
6. 定义描述复数的结构体类型变量，并实现复数之间的加减法运算和输入输出。
7. 已知今天是星期几，输出100天后是星期几。用枚举类型实现。
8. 定义一个描述三种颜色的枚举类型(Red, Blue, Green)，编程输出这三种颜色的全排列结果。

# 第9章 指针和引用

## 9.1 指针和指针变量

### 9.1.1 指针的概念

内存以一个字节（Byte）为一个存储单元，每个存储单元都有一个唯一的编号称为地址。变量的地址是指该变量所在存储区域的第一个字节的地址。在C++中，地址即指针。例如定义三个变量:

```
int a;  float b;  char c;
```

编译器为这三个变量在内存中分配存储空间，如图9-1所示。假定变量a、b、c占用的内存空间的第一个单元地址分别是1048、1044、1040，这三个地址就称为变量a、b、c的指针。

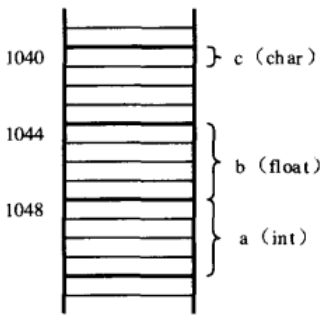



图9-1 变量的存储分配

### 9.1.2 指针变量的定义

指针也是一个数值，C++提供了一种类型的变量用于存放地址值，即存放指针，这种变量就是指针变量。指针变量中存放的是一个内存地址，是另一个变量在内存中的存储位置。

指针变量的定义格式如下:

```
<类型说明符> * <指针变量名>;
```

```
例: int *pi ;      //定义pi为指向 int 类型变量的指针变量
    float *pf ;    //定义pf为指向 float 类型变量的指针变量
    char *pc ;     //定义pc为指向 char 类型变量的指针变量
```

指针类型是一种新的数据类型。指针变量是这种新的数据类型的变量，上述三个指针变量的类型分别是int * 、float * 、char * 。int * 为int型指针类型标识符， float * 为float型指针类型标识符，char * 为char型指针类型标识符。注意: 若pi指向int型量，则称int是pi的基类型。

几点说明:

（1）一个指针变量只能指向同一数据类型的变量，该数据类型是在定义指针变量时明确给定的，例如整型指针变量只能指向整型变量。

（2）C++语言规定有效数据的指针不指向0单元。如果指针变量值为0，即NULL（在ios.h中已定义），表示空指针，即不指向任何变量。

（3）不要把地址值与整数类型值相混淆，例如: 地址2000与整型量2000是两个不同的概念。

### 9.1.3 有关指针的运算符 & 和*

#### 1. & ——取地址运算

功能: 返回变量的内存地址

**例9.1** 使用取地址运算符。

```
int *p , m ;          //定义 p 为指向 int 类型变量的指针
m=200 ;               //将数值200赋给变量m
p=&m ;                //将整型变量 m 的地址值赋给指针变量 p
```

此时，变量在内存中的存储状况如图9-2所示。假定变量m的地址是1040，它的存储空间中存放的是数值200；假定变量p的地址是2000，它的存储空间中存放的是变量m的地址值1040。

**2. * —— 间接访问运算**

功能：访问指针指向的变量

**例9.2** 使用间接访问运算符。

```
int *p , m=200 , n ;
p=&m ;          // p指向整型变量 m
n=*p ;          // 将p指向的变量 m的值取出，并赋给变量 n
*p=100 ;        // 将 100 赋给指针变量 p 所指向的变量 m
```

此处，在已知p指向m的前提下，*p就是访问变量m，即通过p间接访问m。

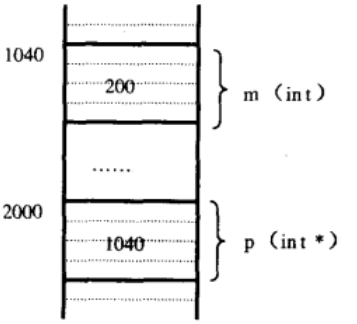



图9-2　指针变量的意义

### 9.1.4 指针变量的初始化

欲使指针指向一个变量，其初始化方法有两种，一是在定义指针变量的同时给它赋值;二是定义指针变量后，用赋值语句给它赋值。如:

```
int a, b ;
int *p1=&a, *p2 ;     // 在定义p1的同时给它赋值，令其指向a
p2 = &b ;             // 在p2定义完成后，用赋值语句给p2赋值，令其指向b
```

### 9.1.5 直接访问和间接访问

在C++中定义的所有变量，编译器都会记录它们的属性，便于以后对它们的访问，如在例9.1中定义的两个变量m和p，编译器记录的属性如表9-1所示。

**1. 直接访问**

在程序中直接使用变量名来存取变量的值称为直接访问方式（也称为直接存取），如m=200。C++内部处理成，从变量属性表中得到变量m的地址1040，通过该地址直接存取内存中变量的值。

**表9-1　变量及其属性**

| 变量名 | 变量类型 | 变量地址 |
|---|---|---|
| m | int | 1040 |
| p | int * | 2000 |

**2. 间接访问**

在程序中通过变量的指针来存取它所指向的变量的值称为间接访问方式（也称为间接存取），如已知p指向m，欲访问m但不直接写出变量名m，而写成*p = 200，此时实际上是将200赋给变量m。C++内部处理成，先从变量属性表中得到变量p的地址2000，取得地址2000中的内容1040，它是变量m的地址，再访问地址1040中的内容，即变量m。

记住：已知int *p , m ; p = &m;，则*p与m等价，p与&m等价。在程序的后面，使用 *p就像使用m一样，对m可以进行的一切操作对*p同样适用，如m = m +1 ，亦可写成 *p = *p + 1; 。同样，使用p就像使用&m一样。

**例9.3** 变量的直接访问和间接访问。

```
#include <iostream.h>
void main( )
{  char c='A' ;
   char *cp=&c ;          // A
   cout << c << *cp << ',' ;
   c='B' ;
   cout << c << *cp << ','  ;
```

```
    *cp='a' ;
    cout << c << *cp << endl ;
}
```

在程序的A行定义指针cp，并给它赋初值，令其指向c。在程序中可以通过变量名c直接访问变量c，也可以通过 *cp 间接访问变量c（c与* cp等价）。该程序输出:

```
AA, BB, aa
```

**例9.4** 变量的直接访问和间接访问。

```
#include <iostream.h>
void main( )
{  int a=1;   float b=5.0;  char c=' A';
   int *p1;   float *p2;    char *p3;
   p1 = &a;  p2 = &b;     p3 = &c;
   cout << a << ', ' << b << ', ' << c << endl;
   cout << *p1 << ', '<< *p2 << ', ' << *p3 << endl;
   *p1 = *p1+1;        // *p1等价于a
   *p2 = *p2+2;        // *p2等价于b
   *p3 = *p3+3;        // *p3等价于c
   cout << a << ', ' << b << ', ' << c << endl;
}
```

运行后，程序输出:

```
1, 5, A
1, 5, A
2, 7, D
```

几点说明:

(1) 指针变量必须通过初始化或赋值获得值后（即必须明确指向某一变量），才可以利用它进行间接访问。

(2) 若有int m ; int *p = &m;

画线部分的意义是将m的地址赋给p，而不是将m的地址赋给p指向的空间。上述两个语句的意义等价于int m ; int *p ; p = &m;

(3) 注意指针变量的指向类型，只有相同类型的数据变量的地址才能赋给该指针变量。即整型变量的地址只能赋给整型指针。如在例9.4中，若出现p1 =&b; ，则是无意义的。

(4) 同类型指针变量之间可以相互赋值，不同类型的指针变量一般不能相互赋值，如在例9.4中，若出现p1 = p2;则是无意义的。

(5) 允许将一个整型常数经强制类型转换后赋给指针变量，如:

```
float *fpp ;  fpp = (float *)5000 ;
```

意义: 将5000作为一个地址值赋给指针变量fpp。

**例9.5** 交换两个指针的指向。

```
#include <iostream.h>
void main( )
{  int x = 10 , y = 20 ;
   int *p1=&x ,*p2 = &y, *t ;
   cout << *p1 << '\t' << *p2 << endl ;
   t = p1;  p1 = p2 ;  p2 = t ;                    // A
   cout << *p1 << '\t' << *p2 << endl;
}
```

此程序的输出是:

```
10   20
20   10
```

初始时，指针p1、p2分别指向x、y，在A行交换它们的指向后， p1、p2分别指向y、x，如图9-3所示。

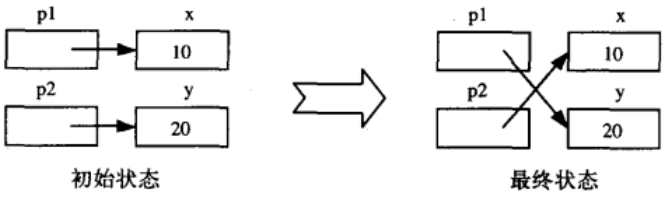



图9-3　交换指针变量的指向

**例9.6**　通过指针，交换两个指针所指向的变量的值。

```
#include <iostream.h>
void main( )
{  int x = 10 , y = 20 ;
   int *p1=&x ,*p2 = &y, t ;
   cout << x << '\t' << y << endl ;
   t = *p1;  *p1 = *p2 ;  *p2 = t ;      // A
   cout << x << '\t' << y << endl ;
}
```

此程序的输出是:

```
10   20
20   10
```

初始时，指针p1、p2分别指向x、y，在A行通过间接访问，交换它们指向的内容后， p1、p2仍然指向x、y，但x、y的内容发生了变化，如图9-4所示。

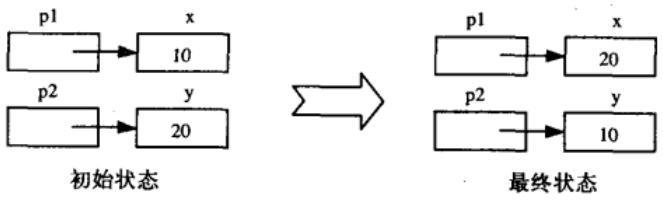



图9-4　通过指针交换它们指向的变量的值

### 9.1.6　地址值的输出

指针的值（即地址值）是可以被输出的。

**例9.7**　输出指针的值。

```
#include <iostream.h>
void main(void)
{    int a, *p1;  float b, *p2;  double d, *p3;
     p1=&a;    p2=&b;   p3=&d;
     cout << "p1=" << p1 << endl;
     cout << "p2=" << p2 << endl;
     cout << "p3=" << p3 << endl;
}
```

默认输出的是十六进制地址值，形如:

```
p1=0x0012FF7C
p2=0x0012FF74
p3=0x0012FF68
```

如果希望输出十进制地址值，则改写程序为:

```
#include <iostream.h>
void main( )
{    int a, *p1;  float b, *p2;  double d, *p3;
     p1=&a;    p2=&b;   p3=&d;
     cout << "p1=" << ( int ) p1 << endl;
     cout << "p2=" << ( int ) p2 << endl;
     cout << "p3=" << ( int ) p3 << endl;
}
```

此时，将地址值强制转换成int型数据，程序的输出形如:

```
p1=1245052
p2=1245044
p3=1245032
```

## 9.2 指针做函数参数

C++中，一种函数参数的传递方式是传值调用，这时参数的传递是单向的，即，函数调用时，将实参的值单向赋值给相应的形参，然后形参和实参就无关了。基本类型量、指针类型量或是构造类型量，都可以使用传值调用方式进行参数的传递。

### 9.2.1 基本类型量做函数参数

**例9.8** 基本类型量做函数参数。

```
#include <iostream.h>
void swap(int x, int y)
{  int t;
   t = x ;  x = y ;  y = t ;
}
void main( )
{  int  x=3, y=9;
   swap(x, y);
   cout<< x<< ',' << y <<endl;
}
```

程序运行后，输出:

```
3, 9
```

此例中，主函数中x和y是主函数内部的局部变量；函数swap( )的形参x和y是在swap( )内部的局部变量，它们与主函数中的x和y虽然同名，但却是不同的变量，占据不同的存储空间。当执行函数调用时，依次将实参x和y的值赋给形参x和y（相当于有int x=x,int y=y;，等号右边为实参x和y），刚进入函数swap( )内部时的内存状况如图9-5a所示，图中虚线箭头表示参数传递的方向。

在swap( )内部，语句依次执行，返回之前，swap( )函数中局部变量x、y的值分别是9、3，函数返回时，系统自动将swap( )函数中的局部变量x、y、t的存储空间撤销，即被调函数的局部变量不存在了。主函数变量中x、y仍然保持其原值，如图9-5b所示。

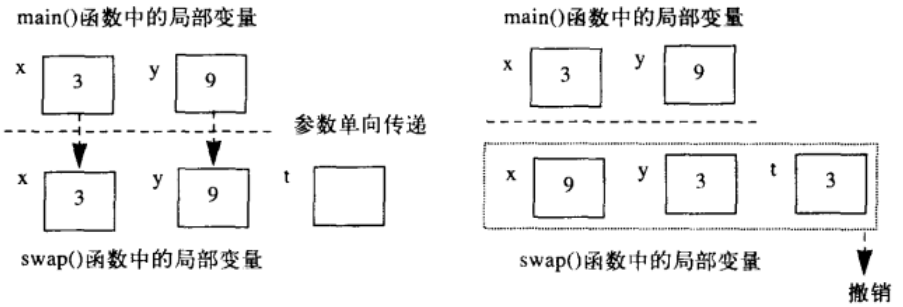



图9-5　函数调用时基本类型参数的传递

### 9.2.2　指针变量做函数参数

**例9.9**　指针做函数参数。

```
#include <iostream.h>
void swap(int *px, int *py)
{  int t;
   t=*px ; *px=*py ; *py= t;
}
void main( )
{
  int  x=3, y=9, *p1, *p2;
  p1=&x; p2=&y;
  swap (p1, p2);
  cout<< x<< ', '<< y<<endl;
}
```

程序运行后，输出:

```
9, 3
```

此例中，主函数中p1和p2是主函数内部的局部指针变量；函数swap( )的形参px和py是在swap( )内部的局部指针变量。当执行函数调用时，依次将实参p1和p2的值赋给形参px和py，相当于做赋值操作“int *px = p1; int *py = p2;”，假定主函数中x、y变量的地址分别是1004、1000，则刚进入函数swap( )时的内存状况如图9-6a所示。

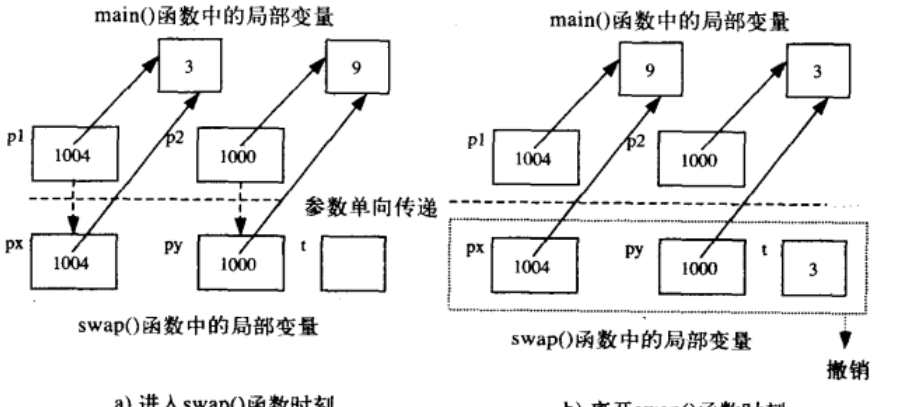



图9-6　函数调用时指针类型参数的传递

在被调函数中通过指针px、py间接访问主函数中的x、y变量，交换x、y的值。当swap( )函数执行结束时，撤销指针变量px、py以及变量t，但主函数中的x、y变量依然存在，且它们的值已经被改变了。返回主函数后，输出的x、y的值分别是9和3。

**例9.10** 约简分数，即用分子分母的最大公约数除分子、分母。

```
#include <iostream.h>
void lowterm(int *num , int *den)
{
   int n , d , r ;
   n = *num ;                          // *num间接访问a
   d = *den ;                          // *den 间接访问b
   while(d != 0 )                      // 用辗转相除法，求分子、分母的最大公约数
   { r = n%d ; n = d ; d = r ; }
   if(n>1)                             // 当最大公约数n大于1时，用n除分子、分母
   {    *num = *num/n ;
        *den = *den/n ;
   }
}
void main(void)
{  int a=14,b=21;                      // a是分子，b是分母
   cout << "分数: " << a << '/' << b << endl ;
   lowterm(&a,&b) ;
   cout << "约简后: "<< a << '/' << b << endl ;
}
```

运行此程序，输出:

```
分数: 14/21
约简后: 2/3
```

调用函数lowterm( )，参数的传递相当于做赋值操作: `int *num = &a; int * den = &b;`。

### 9.2.3 从被调函数中返回信息的几种方法

将被调函数的处理结果返回给主调函数，一般有四种方法。

(1) 利用return语句返回值，例如:

```
int min(int a,int b)
{  return((a<b)?a:b);  }
```

此方法的局限是: 只能返回一个计算结果。

(2) 利用全局变量得到函数调用结果，例如:

```
int max, min ;
void fun(int a, int b)
{ max=(a>b)?a:b ; min=(a<b)?a:b ; }
void main( )
{  fun(5,8) ;
   cout<<max<<','<<min<<endl;
}
```

此方法的好处是可以得到多个计算结果，但安全性欠佳，因为使用了全局变量。

(3) 利用指针变量作为函数参数来取得函数处理的结果，例如:

```
void fun(int a, int b, int *pmax, int *pmin)
{ *pmax=(a>b)?a:b ; *pmin=(a<b)?a:b ; }
void main( )
```

```
{  int a, b, max, min ;
   cin>>a>>b ;
   fun(a, b, &max, &min) ;
   cout<<"max="<< max <<endl<<"min="<< min <<endl ;
}
```

此方法较好，可安全地得到多个结果值。但是指针做参数较难理解，而且书写较麻烦。此方法涉及形参指针变量空间的动态分配。

（4）利用引用作为函数参数来取得函数处理的结果，例如：

```
void fun(int a, int b, int &ma, int &mi)
{ ma=(a>b)?a:b ; mi=(a<b)?a:b ; }
void main( )
{  int a,b,max,min;
   cin>> a>>b ;
   fun(a, b, max, min) ;
   cout<<"max="<< max <<endl<<"min="<< min <<endl ;
}
```

此方法较好，可安全地得到多个结果值。形参变量是实参的别名，不涉及形参变量空间的动态分配，使用起来较方便、直观。

## 9.3　指针和指向数组的指针

### 9.3.1　指针和一维数组

#### 1. 数组名——数组存储区的起始地址

如有定义int a[10];，则a[ ]数组元素在内存中是连续存放的，C++规定，数组名的值是该存储区的首地址，即a[0]元素的地址，是地址常量，而且其地址类型是int *型。例如：

有定义int a[10];

　　float b[10];

如果内存分配如图9-7所示，则a是a[0]元素的起始地址，b是b[0]元素的起始地址，即a与&a[0]等价，b与&b[0]等价。假定a、b两个数组是连续存放的且b数组在前（低地址），a数组在后（高地址），又假定b的值，即b[0]的起始地址是1000，则a的值是1040。

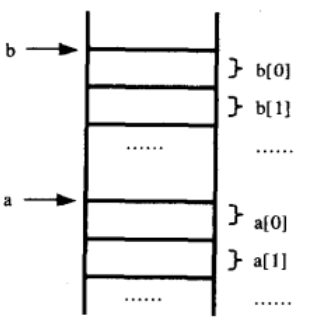



图9-7　一维数组的存储

#### 2. 指向一维数组元素的指针变量

如有定义int a[10]; int *p ;，我们知道数组的每个元素a[i]都是一个int型变量，而p是指向int型变量的指针，那么p可以指向a数组的任意一个元素，如果p=&a[6];，则将a[6]元素的地址赋给了p，p就指向a[6]。当然，也可以将a[0]元素的地址赋给p，即p=&a[0] 或p = a（&a[0]与a等价），这时*p即间接访问a[0]，那么能否通过p间接访问数组a的任意一个元素呢？答案是肯定的。下面首先介绍指针可进行的运算，然后介绍如何通过指针来访问数组的任意一个元素。

#### 3. 指针可进行的运算

指针一般可进行加、减、比较运算。

- **“指针 + 数值”，“指针－数值”的意义**

如果：p 是指针，n 是正整数，则：“p ± n”的意义是，p+n指向它后面第n个元素；p－n指向它前面第n个元素，而“p ± n”的实际值是“p的值 ± n*c”，c是p的基类型占用的存储字节数。即：若p是int型指针，则c等于4；若p是char型指针，则c等于1；若p是double型指针，则c等于8；依此类推。

例: float a[5], *p1 = &a[0], *p2 ;
p2 = p1+3 ;

由图9-8中推论，指针变量加上正整数p+n表示指针向高地址方向移动n个数据存储单元，指向的新地址为: 原地址+sizeof(类型说明符)*n。如上例，若a的值（即a[0]的地址）为 1000，则p1的值为1000，p2的值为1012。

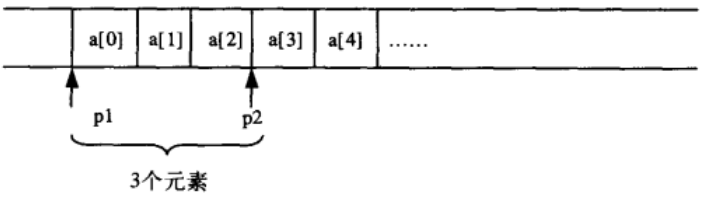



图9-8 “指针+正数”的意义

例: float a[5], *p1 = &a[3], *p2 ;
p2 = p1－2 ;

由图9-9中推论，指针变量减去正整数 p－n表示指针向低地址方向移动n个数据存储单元，指向的新地址为: 原地址－sizeof(类型说明符)*n。如上例，若a的值(即a[0]的地址)为1000，则p1的值为1012， p2的值为 1004。

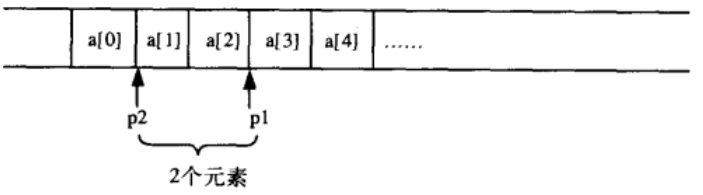



图9-9 “指针－正数”的意义

有定义int a[10]; int *p = a;，如果想访问数组的第i个元素，可写成*(p+i)，或可以直接写成p[i]。因为我们在程序中访问数组第i个元素，一般写成a[i]，此时C++内部处理成: 将a的值加i后，取其指向的内容，实际上处理成*(a+i)，所以如果在程序中书写p[i]，则C++会将其处理成*(p+i)。

**例9.11** 以下程序用指针间接访问数组元素，求出数组元素之和，然后先输出数组全体元素，再输出求和结果。

```
#include <iostream.h>
void main(void)
{ int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }, *p = a, sum=0, i ;
  for( i=0; i<10; i++) sum = sum + *(p+i);
  for( i=0; i<10; i++, p++) // 指针变量可进行自加、自减运算
        cout << *p<<'\t';
   cout << endl << "sum=" << sum << endl;
 }
```

- **“指针－指针”的意义**

指向相同数据类型的指针变量可以相减，其结果为两指针所指向地址之间数据的个数。

例: int *px ,*py , n , a[5] ;
px=&a[1]; py=&a[4];
n = py－px ; // 结果，n值为3。

假定a数组的起始地址是1000，则py的值是1016，px的值是1004，py－px的值不是1016－1004，而

是(1016－1004)/4，即3。指针相减的含义如下：

```
数据个数 = (py地址值 － px地址值) / sizeof(类型说明符)
```

**• 指针的比较运算**

两个指针之间的比较运算有六种，它们是==（恒等）、!=（不等）、<（小于）、<=（小于等于）、>（大于）、>=（大于等于）。比较运算就是直接比较两个地址值的大小。如果相等，则表示两个指针指向同一变量；否则，指向不同变量。运算产生的结果为0或1。

例：有定义int a[5], *px, *py ;，px,py指向如图9-10所示，指针计算如下：

① px= =py　　判断px、py是否指向同一变量，计算结果为0。

② px < py　　若px指向前面的元素，py指向后面的元素，计算结果为1。

③ px= =NULL　　判断px是否为空指针，计算结果为0。

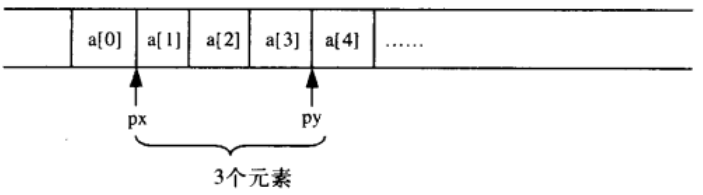



图9-10　“指针－指针”的意义

**例9.12**　改写例9.11，注意指针的比较运算。

```
#include <iostream.h>
void main(void)
{  int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }, *p, sum=0 ;
   for(p = a ; p < a+10; p++) // 此循环结束后，p指向a[9]之后的存储空间
         sum = sum + *p ;
   for(p = a ; p < a+10; p++) // 此循环初始时，需重新给p赋初值
         cout << *p<<'\t';
   cout  << endl << "sum=" << sum << endl;
}
```

**4. 如何获取数组任一元素的地址、任一数组元素的值**

若有定义int a[10], *p ; p=&a[0];，则各指针的指向如图9-11所示。

第i个元素地址的表示为：

- a+i　数组名法
- p+i　指针法
- &a[i]　下标法
- &p[i]　下标法

第i个元素值的表示为：

- *(a+i)　数组名法
- *(p+i)　指针法
- a[i]　下标法
- p[i]　下标法

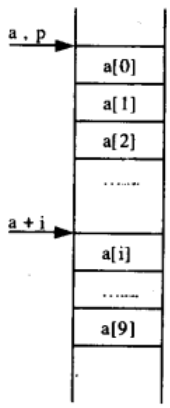



图9-11　一维数组元素指针

通过指针和数组名访问内存方式的区别是：

（1）指针变量是地址变量，其指向由所赋的值确定。

（2）数组名是地址常量（指针常量），恒定指向数组的第0个元素。不允许a++操作。

有了数组的指针表示，前面我们学过的有关数组的算法一般都可以改写成通过数组元素指针间接访

问数组元素。

**例9.13** 将数组元素逆向存放。

```
#include <iostream.h>
void  main(void)
{
      int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, t ;
      int *p1=a, *p2=a+9 ;
      while(p1<p2)
      {  t = *p1; *p1 = *p2; *p2=t; p1++ ; p2 - - ; }
      for(p1 = a; p1 < a+10; p1++)  cout << *p1 << '\t ' ;
      cout << endl ;
}
```

### 9.3.2 一维数组元素指针做函数参数

回忆以前的例子，用函数实现将数组元素逆向存放。

**例9.14** 用函数实现将数组元素逆向存放。

```
#include <iostream.h>
void  reverse( int b[ ], int n )
{  int i=0, j= n-1, t;
   while(i<j )
   { t=b[i]; b[i]=b[j]; b[j]=t; i++; j- -; }
}
void  main(void)
{  int a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, i ;
   reverse(a, 10);
   for(i=0; i<10; i++)  cout << a[i] << '\t ' ;
   cout << endl ;
}
```

在被调函数中数组b[ ]的元素的值被改变后，主函数中对应实参数组a[ ]的元素的值也发生了变化，不需要return语句返回数组的值。而我们知道函数参数的传递是单向的，即将实参值赋给形参，对形参的赋值不影响实参值，但用数组做参数却可以带回数组元素值，这不是违反了参数单向传递的原则了吗？初学C++的人最容易感到困惑，学习了指针以后，这个问题就可迎刃而解了。

让我们看看用数组做函数参数的本质。上述函数调用reverse(a, 10);的第一个实参a是一维数组名，它是数组的起始地址，对应形参的定义形式是int b[ ]，C++规定，形参的这种定义方式即定义了一个指针变量（当然形参也可以定义成int *b），指向整型数，在函数调用参数传递时实际上做了一个赋值操作即b = a ，在被调函数reverse中写出的b[i]形式，C++自动处理成 *(b+i)，即通过指针b间接访问a数组元素。形参定义形式int b[ ]，在方括号中可以写任意整型常量，不影响b的性质，此处方括号中的量只不过是一个形式罢了。

从图9-12中，我们可以更加清楚地认识指针做函数参数的本质。

既然b在reverse ( )函数中是一个指针变量，因而我们可以将reverse ( )函数改写如下:

```
void  reverse( int b[ ], int n )
{  int  t;
   int *p2 = b+n-1;
   while(b < p2)
   {  t = *b; *b = *p2; *p2=t; b++; p2 - - ; }
}
```

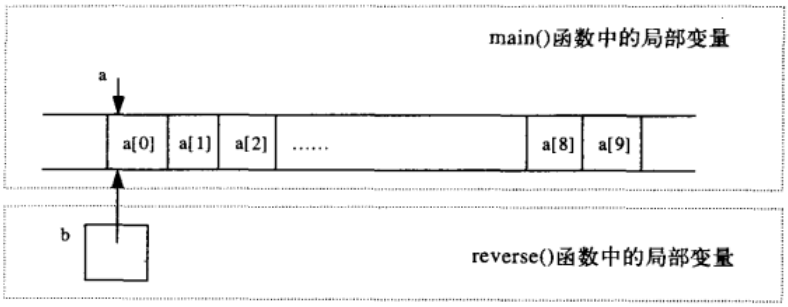



图9-12　指针做函数参数的意义

就本例而言，reverse ( )函数结束时，指针b指向a[5]，即形参b的值已经发生了变化，因为b是一个局部指针，函数返回时并不影响实参a的值，a仍然指向a[0]。

若欲将数组的起始地址传递给被调函数的指针形参，相应实参和形参的写法有多种，在被调函数中访问实参数组元素的方式也有多种:

```
void fun(int b[ ], int n) //或void fun(int b[10], int n) 或void fun(int *b, int n)
{ ......
   b[i]                    //或 *(b+i) 或 *b++ 等
  ......
 }
void main(void)
{  int  a[10];
   int *p = a;
   ......
   fun(a, 10);             //或 fun(p, 10) 或 fun(&a[0], 10)
   ......
}
```

有了地址传递的概念，我们可以编制出许多灵活的程序，如将例9.14的主函数改写后，可实现将数组a的部分元素逆向存放，程序如下:

```
void  reverse( int b[ ], int n ) //被调函数不变
{   int i=0, j=n-1, t;
    while(i<j )
    { t=b[i]; b[i]=b[j]; b[j]=t; i++; j- -; }
}
void  main(void)
{  int  a[10]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, i ;
   reverse(a+3, 6);
   for(i=0; i<10; i++)  cout << a[i] << '\t ' ;
   cout << endl ;
}
```

实参指针a+3指向a[3]，函数调用时，将a+3赋给b，在被调函数中实现将从a[3]开始的6个元素逆置，所以此程序运行后输出:

```
1    2    3    9    8    7    6    5    4    10
```

**例9.15**　分别求数组前10个元素和后10个元素之和。

```
#include <iostream.h>
int fsum(int *array, int n)
```

```
{   int i, s=0 ;
    for(i=0; i<n; i++)
        s+=array[i] ;
    return(s);
 }
void main(void)
{  int a[15]={ 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 } ;
   int shead, stail ;
   shead = fsum(a, 10) ;                    // 第1次调用
   stail = fsum(&a[5], 10) ;                // 第2次调用
   cout << shead << ', ' << stail << endl ;
}
```

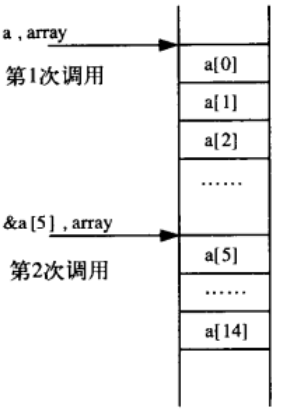



图9-13 元素指针做参数

主函数第1次调用fsum( )函数时，第1个参数传递的是a[0]的地址，那么array指向a[0]，在fsum( )函数中求出的是从a[0]（含a[0]元素）开始的10个元素的和。主函数第2次调用fsum( )函数时，第1个参数传递的是a[5]的地址，那么array指向a[5]，在fsum( )函数中求出的是从a[5]（含a[5]元素）开始的10个元素的和（见图9-13）。

**例9.16** 求出数组元素中的最大值和最小值。

```
#include <iostream.h>
void max_min_value (int *array, int n, int *maxp, int *minp)
{                                       // maxp指向主函数max变量，minp指向主函数min变量
   int *p, *array_end;
   array_end = array + n ;
   *maxp=*minp=*array ;                 // 假定第0个元素既是最大值又是最小值
   for(p=array+1; p < array_end; p++)
   {   if( *p>*maxp ) *maxp=*p;
       else if( *p<*minp ) *minp=*p;
   }
 }
void main(void)
{
   int i, number[10], *p = number, max, min;
   for(i=0; i<10; i++)
        cin >> *( p+i ) ;               // 输入数组元素值
   max_min_value( p, 10, &max, &min );
   for(i=0; i<10; i++)
        cout << *( p+i ) << '\t ' ; // 输出数组元素值
   cout << "max value = " << max << "min value = " << min << endl ;
}
```

本例涉及数组指针做参数（第1个参数p）以及基本类型变量的指针做参数（第3、4个参数&max，&min）。在介绍函数时，我们知道可以通过函数的返回值（return）带回一个运算结果，学习了指针做参数后，可将多个变量的指针传递给被调函数，在被调函数中通过指针间接访问主函数变量，改变主函数变量的值，从而得到多个结果。

**例9.17** 在函数中通过指针访问数组元素，实现选择法排降序。

```
#include <iostream.h>
void sortd (int *a, int n)
{
   int *p, *q, *maxp, t;
```

```
    for( p=a ; p< a + n -1; p++)
    {    maxp = p ;
         for( q=p+1; q< a + n; q++)
              if( *q>*maxp ) maxp = q;        // 排降序
         if( maxp!=p )
         { t= *p ; *p=*maxp; *maxp=t ;  }
    }
 }
void main(void)
{
   int i, a[10] ;
   for(i=0; i<10; i++)
       cin >> *( a+i ) ;                      // 输入数组元素值
   sortd( a, 10 );
   for(i=0; i<10; i++)
      cout << *( a+i ) << '\t';               // 输出排序后的数组元素值
}
```

### 9.3.3 指针和字符串

#### 1. 字符数组和字符指针

在C++中，我们可以用字符数组存放字符串。也可以定义一个指针指向一个字符串常数。

**·定义一个字符数组存放字符串**：

```
char str[10]= "Hello!";
```

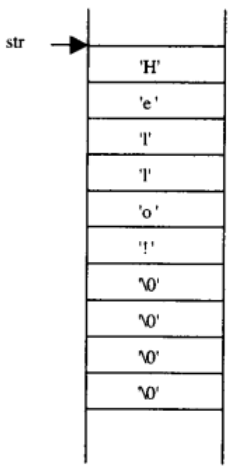



图9-14　字符串存储在字符数组中

str是一维字符数组名，即数组的起始地址。编译器给数组分配10个字节的存储空间，将字符串的值放入该存储空间。实际上只使用了数组前7个字节的空间，后3个字节暂时无用。内存空间的分配如图9-14所示。

（1）字符数组输入／输出：

```
cin >> str;                 // 不可输入带空格的字符串
cin.getline(str, 80);       // 可输入带空格的字符串
cout << str;                // 输出字符串
```

（2）字符数组赋值：`strcpy(str, "ABCD" );`。

（3）访问第i个字符str[i] 或 *(str+i)。

（4）获取第i个字符的地址 &str[i] 或 str+i。

字符串存放在一维数组中，str是起始地址。可以看出，在字符串的输入、输出及字符串处理函数中，使用的都是数组名。使用它与一般的一维指针一样。在C++中，处理字符串时，将从起始地址开始到字符串结尾标志 '\0' 之间的内容作为整体看成一个字符串。

例如：
```
char str[10]= "Hello!";
strcpy(str+2, "ABCD")
cout<<str;
```

则输出为HeABCD。str+2代表字符串 "llo!"。

**·定义一个字符指针并令其指向一个字符串**：

```
char *strp= "Hello!" ;
```

"Hello!"是一个字符串常数，在内存中占据一片连续的存储区，C++将"Hello!"的值处理成该字符串

的存储区的首地址。所以，上述语句的含义是将字符串"Hello!"的首地址赋给指针变量strp。等价于：char *strp ; strp= "Hello!";。

注意字符数组名（str）与字符指针（strp）的区别：

（1）str是指针常量，不论是否对字符串赋值，数组空间已分配，str的指向明确。不允许给str重新赋值，如str= "Hello!"是错误的。而strp是指针变量，若不赋初值，则其指向不确定。

（2）给字符数组赋字符串值：strcpy(str,"ABC");，str的指向不变，改变的是存储单元的内容。也就是说，将字符串常数"ABC"复制到str指向的数组空间中。给字符指针赋值：strp="Hello!"; 表示使strp指向字符串常量"Hello!"。而strp=str; 表示将strp指向字符串数组str的第一个字符。

**例9.18** 要求自己编写代码，将字符串b的内容追加到字符串a的尾部，不能调用系统函数strcat( )。

```
#include <iostream.h>                    // 解1: 用数组元素访问方式实现
#include <string.h>
void main( )
{  char  a[20]="ABCD", b[10]="EFG";
   int  i, j;
   i = strlen(a);                        // i是字符串a[ ]结尾'\0'的下标
   for(j=0; b[j]!='\0'; i++, j++)
       a[i]=b[j];
   a[i]='\0';
   cout << a << endl ;
}

#include <iostream.h>                    // 解2: 用字符指针访问方式实现
void main( )
{ char  a[20]="ABCD", b[10]="EFG";
  char *pa=a, *pb=b;                     // pa指向a[0]、pb指向b[0]
  while(*pa!='\0')  pa++;                // 循环结束后，pa指向a[ ]尾部的'\0'
  while(*pb!='\0')
  { *pa=*pb;  pa++; pb++; }
  *pa='\0';
  pa=a;
  cout << pa << endl ;
}
```

### 2. 字符串指针做函数参数

与一维整型数组元素的指针类似，字符串指针也是一维数组元素的指针。字符串指针指向的是char类型的变量（元素）。可以做函数参数。字符串指针做实参，传递的是字符串的首地址。

**例9.19** 字符串指针做函数参数。

```
#include <iostream.h>
int my_strlen(char *s)                   // 自定义函数，实现求字符串的长度
{   int n;
    for(n=0; *s!='\0'; s++)              // 循环结束后，s指向'\0'
            n++;
    return(n);
}
void main( )
{  char str[ ]="Hello!";
   cout << *str << '\t';                 // A
   cout << my_strlen(str) << '\t';       // B
```

```
    cout << *str << endl;                    //C
}
```

程序的运行结果是:

```
H          6          H
```

在A行，由于str是字符串的首地址，而且它是一个char * 类型的地址，因此 *str是数组第0个元素的值，即str[0]，所以输出的是H。经常会误认为*str的值是"Hello!"。现在，与整型数组作个比较。如有int a[10]={1, 2};，此时*a就是指a[0]，而不是整型数组整体。在B行，调用函数my_strlen( )给出的实参是str，是一个地址，赋给形参指针变量s，相当于char *s = str。而形参s是函数my_strlen( )内部的局部指针，所以，即使在被调函数中s的值发生了变化，也不影响实参str的值。在C行仍然输出H。

**例9.20** 比较两个字符串的大小。

```
#include <iostream.h>
int my_strcmp(char *s, char *t)          //自定义函数
{ for( ; *s==*t; s++, t++)
      if(*s=='\0') return(0);
  return(*s-*t);
}
void main( )
{  char s1[ ]="Hello!";
   char *s2="Hello!";
   cout<< my_strcmp(s1, s2)<<'\t';
   cout<< my_strcmp("Hi", s2)<<endl;
}
```

函数my_strcmp( )的返回值的意义与字符串处理标准库函数strcmp( )的意义相同。程序的输出结果为:0 4。请读者自行分析输出结果。

**例9.21** 请阅读并理解下述若干函数，它们的功能都是实现字符串复制。

```
#include <iostream.h>
void copy_string(char from[ ], char to[ ])
{  int i=0 ;
   while(from[i]!='\0')
     { to[i]=from[i]; i++ ; }
   to[i]='\0' ;
}
void copy_string(char from[ ], char to[ ])
{  int i=0 ;
   while((to[i]=from[i])!='\0') i++ ;
}
void copy_string(char *from, char *to)
{  while((*to=*from)!='\0')
   {  to++ ; from++ ; }
}
void copy_string(char *from, char *to)
{  while((*to++ = *from++)!='\0') ;  }
void copy_string(char *from, char *to)
{  while(*to++ = *from++) ;  }
```

以上五个函数的功能是一样的，可以用下述主函数调用。

```
void main( )
{  char s1[20], s2[20];
```

```
    cin>>s1;
    copy_string(s1, s2);
    cout<<s2<<endl;
}
```

### 9.3.4 多维数组指针（二维数组）

#### 1. 二维数组的地址

有定义int a[3][4]；，数组中的全体元素如下(见图9-15):

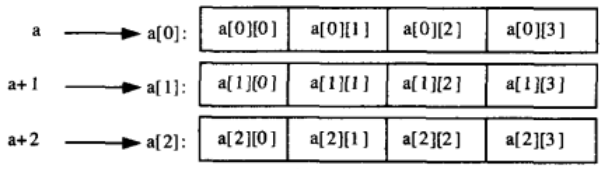



图9-15 二维数组的构成

在C++中，可以这样来理解二维数组，将数组的每一行看成是一个“大元素”，分别是a[0]、a[1]和a[2]，二维数组是由这些“大元素”构成的一维数组。二维数组名a是这些“大元素”的指针。a指向a[0]，a+1指向a[1]，a+2指向a[2]。而这些“大元素” 本身又是含有4个整型元素的一维数组（见图9-16）。

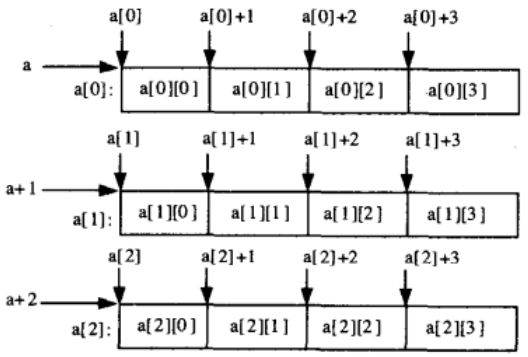



图9-16 二维数组行指针和元素指针示意

以第0行为例，a[0]是第0行一维数组的数组名，a[0]指向元素a[0][0]，a[0]+1指向元素a[0][1]，依次类推。

**• 行指针和元素指针**

a和a[i]是两种类型不同的指针，a是行指针，a+1移动一个“大元素”(即一行元素)，a+i是第i行的行指针。a[i]是元素指针，指向第i行一维数组起始元素，其类型是int *。通过指针a[i]可以存取一个具体的元素。在C++的处理中，*(a+i) 等价于a[i]。所以可以说在行指针之前加 * 得到元素指针。

**• 二维数组中元素a[i][j]地址的表示**

有了上述行指针和元素指针的概念，我们可以用下述方法表示元素a[i][j]的地址：&a[i][j]、a[i]+j和*(a+i)+j 。

**• 二维数组中元素a[i][j]的值的引用方法**

有四种方法可以访问数组元素a[i][j]的值：a[i][j]、*(a[i]+j)、*(*(a+i)+j) 和 (*(a+i))[j] 。

**例9.22** 以下程序输入二维数组的值，并求出二维数组的全体元素之和。

```
#include <iostream.h>
void main( )
```

```
{  int a[3][4];
   int sum=0, i, j;
   for(i=0; i<3; i++)
        for(j=0; j<4; j++)
        cin>> *(a[i]+j);            //A
   for(i=0; i<3; i++)
       for(j=0; j<4; j++)
            sum += *(*(a+i)+j);     //B
   cout << sum << endl;
}
```

程序中A行和B行用了两种通过指针访问二维数组元素的方法，当然也可以写成a[i][j]或(*(a+i))[j]。

### 2. 指向一维数组的指针变量（行指针）

我们在前面讲了，a和a[i]是两种类型不同的指针，并且指出a[i]的类型是 int * 。而a是指向“大元素”的指针，这个“大元素”是一维数组，我们看看a的类型是什么？

若二维数组的每一行有4个元素，定义一个指向具有4个元素的一维数组的指针变量（即行指针变量）的方法是int(*p)[4];。这样定义后，p的特性与a的特性一样（int a[3][4];），它们的类型都是int(*)[4]。于是可以进行赋值 p=a 。需要指出的是，a是指针常量，而p是指针变量。也可以这样赋值p=a+2，此时p指向a数组的第2行（a数组从第0行开始）。若有p=a，则通过p可以用四种方法访问a数组元素a[i][j]，分别是p[i][j]、*(p[i]+j)、*(*(p+i)+j)和(*(p+i))[j]。

**例9.23** 改写例9.22，通过行指针变量访问数组a的元素。

```
#include <iostream.h>
void main( )
{  int a[3][4], (*p)[4];
   int sum=0, i, j;
   p=a;
   for(i=0; i<3; i++)
       for(j=0; j<4; j++)
          cin>> *(p[i]+j);              //A
   for(i=0; i<3; i++)
       for(j=0; j<4; j++)
          sum += *(*(p+i)+j);           //B
   cout << sum << endl;
}
```

同样地，程序中A行和B行对二维数组元素的访问也可以写成p[i][j]或(*(p+i))[j]。

### 3. 二维数组名做函数参数

**例9.24** 改写前例，用二维数组名做函数参数，在被调函数中求二维数组全体元素之和。

```
#include <iostream.h>
int total( int(*p)[4], int n )  //A
{   int i, j, sum=0;
    for( i=0; i<n; i++)
           for( j=0; j<4; j++)
                  sum += *(*(p+i)+j) ;
    return sum;
}
void main(void)
{  int a[3][4], sum, i, j;
```

```
    for(i=0; i<3; i++)
        for(j=0; j<4; j++)
            cin>> *(a[i]+j);
    sum = total( a, 3 );
    cout << sum << endl;
}
```

注意在A行，二维数组做参数时形参p的定义形式。另外，第2个参数n为二维数组的行数。在被调函数中，p仅仅是“一个”行指针变量，它的初值为a的值，即相当于在参数传递时有int(*p)[4]=a;。为了体会p是一个指针变量，我们还可将上例的total( )函数做如下改写:

```
int total(int(*p)[4], int n)                    //n 行数
{   int i, j, sum=0 ;
    for( i=0; i<n; i++, p++ )                   //A
        for( j=0; j<4; j++)
            sum += *( *p + j );                 //B
   return sum;
}
```

注意A行和B行。在A行，指针变量p加1，指向下一行。在B行，指针变量p前加*，变成元素指针 *p。通过元素指针 *p 访问一行元素。

注意，二维数组名做参数时，形参有三种写法，无论使用何种写法，其意义都是一样的，传递的是二维数组的行指针。如下所示:

```
void fun( int  b[3][4] )    //形参还可写成 int b[ ][4] 或 int (*b)[4]
{ ……
   *(*(b+i)+j)              //可以通过指针形式访问数组元素
   b[i][j]                  //也可以通过下标形式访问数组元素
  ……
}
void main( )
{  int  a[3][4];
   ……
   fun(a);                  //实参只需写数组名
   ……
}
```

### 4. 将二维数组看成一维数组访问

- **通过元素指针访问二维数组元素** 我们知道，二维数组从逻辑上看是二维的，实际在内存中的物理存放是一维的，即线性的。因此我们可以通过元素指针扫描物理的一维数组，来访问二维数组全体元素。

**例9.25** 输出二维数组全体元素的值。

```
#include <iostream.h>
#include <iomanip.h>
void main(void)
{
  int a[3][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}, *p ;
  for(p=a[0]; p<a[0]+12; p++)                    //A
  {
        if((p-a[0])%4==0) cout << endl ;         //B
        cout << setw(4) << *p ;
```

```
    }
    cout << endl ;
}
```

注意，程序中p是整型指针，在A行，将元素指针a[0]赋给p，p就指向了物理的一维数组的第0个元素，二维数组共有12个元素，最后一个元素的地址是a[0]+11。通过p扫描该一维数组，达到访问二维数组全体元素的目的。另外注意B行指针相减的含义。

- 已知一个二维数组为N行×M列，即a[N][M]，将二维数组存放成一维数组后，求a[i][j] 存放在物理一维数组的第几个位置？如图9-17，二维数组按行存放成物理的一维数组后，图中阴影部分元素放在二维数组任意元素a[i][j]之前，共有 i*M+j 个元素。若序号从0开始，则a[i][j]的序号是i*M+j。所以，若有int *p=a[0];，p是整型指针，指向物理一维数组的第0个元素，而p+i*M+j 是a[i][j]的地址，于是可以通过p和i、j计算出任意元素的地址，从而访问二维数组的任意元素，即 *( p+i*M+j)是a[i][j]的值。

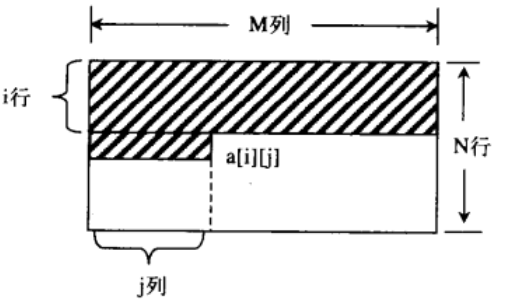



图9-17　元素a[i][j]存放位置示意

访问数组元素a[i][j]时，C++内部处理成：将二维数组看成一维数组，按公式 p+i*M+j 计算出a[i][j]在一维数组中的起始地址，然后存取a[i][j]。a[i][j]的地址计算公式 p+i*M+j 只与二维数组的首地址、i、j 以及列数M有关，而与N无关，所以当使用二维数组名做参数时，形参的书写形式为 int b[ ][M]（或int (*b)[M]），行数可以缺省，但列数M不可以缺省。

**例9.26**　输入二维数组任一行号i，任一列号j，输出其元素a[i][j]的值。

```
#include <iostream.h>
void main(void)
{  int a[3][4]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
   int *p, i, j;
   p=a[0];
   cin >> i >> j;
   cout<< "a[" << i << "][" << j<< "]=" << *(p+i*4+j) << endl;
}
```

此程序运行时，输入 1 3<Enter>，输出 a[1][3]=8。

### 5. 指针类型小结

前面我们学习了一维数组元素指针以及二维数组行指针，下面以整型类型为例将两种指针的意义以及指针本身的类型作一个小结：

（1）int *p; p为一般指针，指向一个整型元素，其类型与一维数组名等价。如有：int a[10]; p=a;，则可用p[i]访问数组元素。

（2）int (*p)[M]; p指向含有M个元素的一维数组，可与每行含有M个元素的二维数组名等价，p 是一个行指针。如有：int a[4][M]; p=a;，则可用p[i][j]访问数组元素。

**表9-2　不同指针形式及对应数据类型**

| 指针表示形式 | 相应的数据类型 |
|---|---|
| a | int(*)[M] |
| *a | int* |
| **a | int |
| &a[i] | int(*)[M] |
| *(&a[i]+j) | int* |

表9-2中列出了几个不同的指针表示形式及其指针类型，帮助读者对指针类型的理解。

## 9.4 指针数组

### 9.4.1 指针数组的定义和使用

#### 1. 基本概念

若数组元素为整型量，我们称之为整型数组；若数组元素为字符型量，我们称之为字符数组。那么，若数组元素为指针，我们就称之为指针数组。指针也是有类型的，如果指针数组中每个元素为整型指针，则为整型指针数组。同样，还有float型指针数组、double型指针数组、char型指针数组等。

#### 2. 指针数组的定义

其定义形式为：<类型名> *<指针数组名>[<元素个数>]

如int *p[10];，表示定义了一个整型指针数组p[10]，它有10个元素p[0]，p[1]，p[2]，...，p[9]，每个元素均为整型指针。

例，程序片段如下：

```
int a=6, b=8, c=2;
int *p[10];
p[0]=&a;   p[1]=&b;  p[2]=&c;                                    //A
cout << *p[0] << '\t' << *p[1] << '\t' <<*p[2] << '\t' <<endl;  //B
```

输出：6　8　2。

说明：A行表示给p数组的前三个元素赋值，即将三个指针值赋给数组的前三个元素。B行中，*p[0]表示间接访问p[0]指向的值，即变量a的值；同理可知*p[1]、*p[2]的意义。

同理：

```
float *pf[10];    //定义了一个float型指针数组，数组共有10个float型指针。
double *pd[5];    //定义了一个double型指针数组，数组共有5个double型指针。
char *pc[20];     //定义了一个char型指针数组，数组共有20个char型指针。
```

**例9.27** 利用指针数组输出另一个一维数组中各元素的值。

```
#include <iostream.h>
void main( )
{    float a[5]={2, 4, 6, 8, 10};
     float *p[5]={ &a[0], &a[1], &a[2], &a[3], &a[4] };  //A
     int i;
     for(i=0; i<5; i++) cout << *p[i] << '\t';
     cout << '\n';
}
```

此例中有两个数组，一个float型数组a[5]，另一个是float * 型数组p[5]。注意，A行表示给指针数组赋初值。指针数组p[5]各元素的指向关系如图9-18所示。

程序运行输出：2　4　6　8　10。

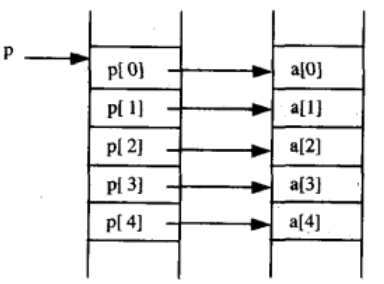



图9-18 指针数组元素的意义（一）

### 9.4.2 使用指针数组处理二维数组

例：
```
int a[3][3], *p[3] ;
for(int i=0; i<3; i++)
     p[i]=a[i];          //A
```

说明：a[3][3]是二维数组，通过前面的学习我们知道，a[0]、a[1]、a[2]分别是每一行元素的首地址，指针类型是 int *，即整型指针。p[3]是指针数组，它的三个元素分别是指向整型变量的指针（元素指针）。上述A行表示将a数组每行元素的起始地址赋给指针数组的相应元素。指针的指向关系

如图9-19所示。

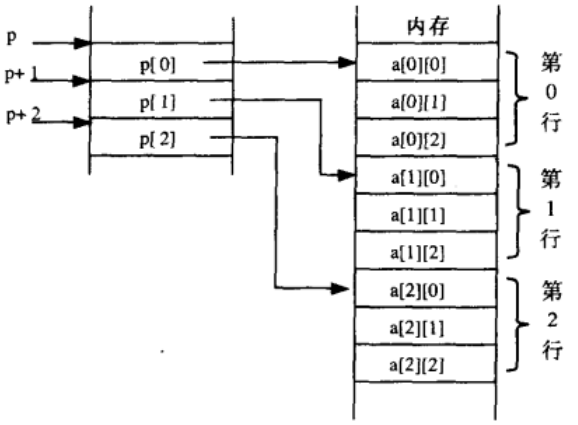



图9-19　指针数组元素的意义（二）

于是通过指针p引用a数组元素的方式有以下四种:

```
p[i][j],  *(p[i]+j),  *(*(p+i)+j), (*(p+i))[j]
```

**例9.28**　输出二维数组全体元素的值。

```
#include <iostream.h>
void main(void)
{  int a[2][3]={{1, 2, 3}, {4, 5, 6}};
   int *pa[ ]={a[0], a[1]}; //通过初始化的方式给指针数组赋初值。
   int i , j ;
   for(i=0; i<2; i++)
   {  for(j=0; j<3; j++)
         cout << *(pa[i]+j) << '\t';
      cout << '\n';
   }
}
```

此程序的输出结果是:

```
1   2   3
4   5   6
```

### 9.4.3　利用字符指针数组处理字符串

通过前面的学习我们知道，C++把字符串常数的值处理成其内存地址，现有定义:

```
char *name[ ]={"George", "Mary", "Susan", "Tom", "Davis"};
```

name是一个指针数组，根据括号中的字符串初值个数可知，共有5个指针，分别指向5个字符串常数，其内存指向的逻辑关系如图9-20所示。

**例9.29**　按字典序将上述5个字符串排成升序，并输出排序后的结果。此程序采用选择法排序，通过交换指针的指向来达到排序的目的。

```
#include <iostream.h>
#include <string.h>
void main( )
{
        char *name[ ]={"George", "Mary", "Susan", "Tom", "Davis"};
```

```
        char *ptr;
        int i, j, k, n=5;
        for(i=0; i<n-1; i++)
        {
              k=i;
              for(j=i+1; j<n; j++)
                    if( strcmp(name[k], name[j])>0 )      // 字符串比较
                          k=j;                            // 记住最小字符串指针的下标
              if(k!=i)
              { ptr=name[i]; name[i]=name[k]; name[k]=ptr; }  // 交换指针数组元素值
        }
        for(i=0; i<n; i++)                                // 输出结果
              cout << name[i] <<'\t';
        cout<<endl;
}
```

在程序输出结果之前，内存各指针的指向关系如图9-21所示。

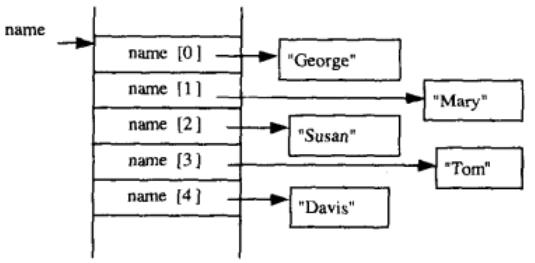



图9-20 字符型指针数组元素的意义

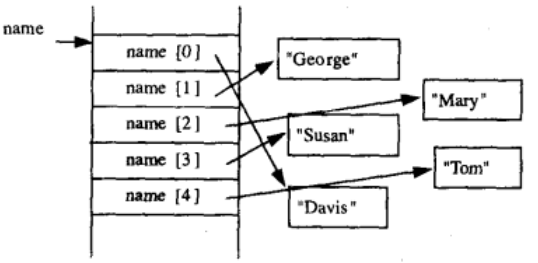



图9-21 排序后字符型指针数组元素的指向

程序的输出是:

```
Davis   George  Mary    Susan   Tom
```

**例9.30** 改写例9.29，学习如何使用指针数组做函数参数。

```
#include <iostream.h>
#include <string.h>
void main( )
{   void sort(char *[ ], int), print(char *[ ], int);   // 函数原型说明，因为函数定义在后
    char *name[ ]={"George", "Mary", "Susan", "Tom", "Davis"};
    int n=5 ;
    sort( name, n);
    print( name, n);
}
void sort(char *name[ ], int n)                          // 函数定义，注意参数的写法
{   char *ptr; int i, j, k;
    for(i=0; i<n-1; i++)
    {
          k=i;
          for(j=i+1; j<n; j++)
                if( strcmp(name[k], name[j])>0 ) k=j;
          if(k!=i)
          {     ptr=name[i];name[i]=name[k];name[k]=ptr;       }
    }
```

```
}
void  print(char *name[ ], int n)                    // 函数定义，注意参数的写法
{  for(int i=0; i<n; i++)
         cout <<name[i] << endl;
}
```

### 9.4.4 指针数组做 main 函数的参数

通常main函数的定义形式如下，它是无参函数:

```
void main( )
{  ......   }
```

实际上，main函数是可以带参数的。带参数的main函数的定义形式为:

```
void main( int argc, char *argv[ ] )
{  ......   }
```

main函数带参数的意义是什么呢？可以通过例9.31来学习main函数参数的意义。

**例9.31** 了解main函数参数的意义。

```
//Li0931.cpp
#include<iostream.h>
void main(int argc, char *argv[ ])
{
        cout<<"argc="<<argc<<endl;
        cout <<"Command name=" << argv[0] << endl;
        for(int i=1; i<argc; i++)
              cout<<argv[i]<<endl;
}
```

上述源程序文件名是Li0931.cpp，假定它被存放在 D:\cpp文件夹下，编译连接后产生可执行程序，文件名为Li0931.exe，它在D:\cpp\Debug下。执行该程序，实际上就是操作系统调用这个程序。操作系统在调用Li0931.exe时，可以给它的主函数传递必要的参数，以使主函数根据不同参数做不同的工作。操作系统如何将参数传递给main函数呢？有以下两种方式:

**(1) 通过DOS命令行参数。**

进入Windows的DOS方式，改变当前工作目录为D:\cpp\Debug，在DOS提示符下键入命令行，形式如下:

```
D:\cpp\Debug> Li0931  par1  par2  /p  /w  <Enter>
```

带波浪下划线的部分称为DOS命令行，操作系统分析该命令行，实际上就是分析字符串" Li0931 par1 par2 /p /w "，将其分解成若干子串，这些子串是由空白字符分隔开的。它们是"D:\cpp\Debug\Li0931.exe"、"par1"、"par2"、"/p"和"/w"，第一个子串比较特别，因为它是命令（可执行程序），操作系统自动地将它所在的路径及其扩展名加上。这些信息被传递给了main函数。main函数参数的意义是，argc是子串的个数（如上例为5），而参数argv是指针数组，各元素的指向如图9-22所示。

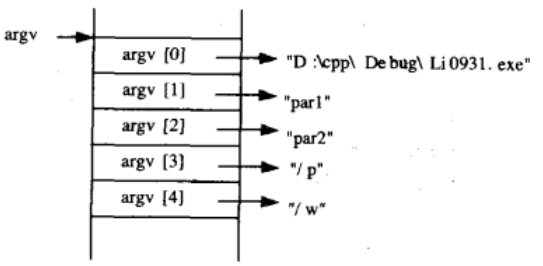



图9-22　main函数argv参数的意义

因此，该程序运行后的结果如下:

```
argc=5
Command name=D:\cpp\Debug\Li0931.exe
par1
par2
/p
/w
```

**(2) 在集成环境中设置命令行参数。**

在集成环境中也可以设置命令行参数以测试主函数参数的正确性。方法是，将上述程序编译连接后，在集成环境中通过Project菜单的Settings命令的Debug标签中的Program arguments编辑行（Project | Settings | Debug | Program arguments)。如输入命令行参数 "par1 par2 /p /w"，注意不需要输入命令名本身Li0931，另外也不要输入双引号。然后运行该程序，结果同上。

用户在编写主函数时，就可以根据不同的参数完成不同的功能。

## 9.5 指向指针的指针

在C++中定义一个基本类型变量时，该变量占用若干存储单元，第一个存储单元的地址为该变量的起始地址，我们可以定义一个指针变量存放该起始地址。同样地，当定义一个指针变量时，该指针变量也占用若干存储单元，也有起始地址，我们也可以定义一个变量存放指针变量的起始地址。存放指针变量起始地址的变量，称之为指向指针的指针，其定义格式为:

```
<类型说明符> ** <指针变量名>
```

例如:

```
int x=3, *p1, **p2;
p1 = &x ;
p2 = &p1 ;
```

假定变量x的地址是1000，变量p1的地址是2000，则内存中的存储状况如图9-23所示。

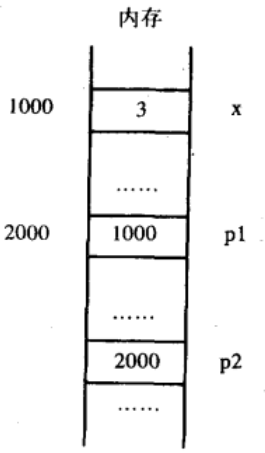



图9-23 指向指针的指针的意义

这时，可以通过p1一级间接访问x，即*p1等价于x。也可以通过p2二级间接访问x，即**p2等价于x。

**例9.32** 定义一个指向指针的指针，用它指向指针数组。

```
#include<iostream.h>
void  main( )
{  char **p ;
   char *s[ ]={ "up", "down", "left", "right" };
   int  i ;
   p=s ;
   for( i=0; i<4; i++ )
        cout << *p++ <<'\t';     //A
   cout<<endl;
}
```

指针数组s有四个元素，它们是四个字符串常数的起始地址，是char * 类型的指针。p指向s数组。A行的 *p++ 就是 *(p++)。A行等价于cout<<*p<<endl; p++;。初始时p指向s[0]，*p就是s[0]，于是输出字符串"up"，然后p加1，指向下一个字符串"down"。运行程序后，输出:

```
up      down    left    right
```

## 9.6 指针和函数

### 9.6.1 函数指针

**1. 函数指针的定义和说明**

一个函数被编译连接后生成一段二进制代码，该段代码的首地址，称为函数的入口地址。C++将函数名的值处理成函数的入口地址。可以定义一个指针变量，使之存放函数的入口地址，我们称这种变量为函数指针变量，简称函数指针。

函数指针的定义格式为:

```
<类型说明符> (<*函数指针>)(<参数类型表>);
```

例如:

`int (*fp)(int, int);` 定义函数指针fp，它应指向具有两个整型参数且返回值为整型数据的函数。

`int max(int, int);` 说明函数max。

`fp=max ;` 将函数max的入口地址赋给指针变量 fp，fp和max都指向函数的入口。注意fp和max的类型是一样的。

几点说明:

(1) 函数指针指向程序代码区，一般数据变量指针指向数据区。

(2) 在定义int (*fp)(int, int)中，注意运算符( )优先级高于*，*fp 两侧括号不能省略。若省略就是另外的含义，参见9.6.2节。

(3) 函数指针不能进行 ++、--、+、- 等运算，因为在程序执行阶段函数代码区不会变动。

### 2. 函数指针的使用

**• 用函数指针调用函数。**

函数可以通过函数名调用，也可通过函数指针调用。如上例fp=max，则对函数max的调用方式可以是:

```
max(实参表);                //通过函数名调用，建议使用
(*max)(实参表);             //通过函数名调用
fp(实参表);                 //通过函数指针调用，建议使用
(*fp)(实参表);              //通过函数指针调用
```

**例9.33** 通过函数指针调用函数。

```
#include<iostream.h>
void main( )
{  int max(int, int), min(int, int);    //函数原型说明
   int (*fp)(int, int);                 //定义函数指针 fp
   int a, b, c, d;
   cin >> a >> b ;
   fp=max;                              //fp 指向 max( ) 函数
   c=fp(a, b);                          //通过 fp 调用 max( ) 函数
   fp=min;                              //fp 指向 min( ) 函数
   d=(*fp)(a, b);                       //通过 fp 调用 min( )函数
   cout<<"max="<<c<<'\t'<<"min="<<d<< endl;
}
int max(int x, int y)  {  return(x>y?x:y);  }
int min(int x, int y)  {  return(x<y?x:y);  }
```

**• 函数指针作为函数参数，实现两个或多个函数之间的调用控制。**

**例9.34** 函数名做参数，通过一个公用接口调用三个函数。

```
#include<iostream.h>
void main( )
{ int max(int, int), min(int, int), add(int, int);   //函数原型说明
  int process(int, int, int (*)(int, int));          //函数原型说明
  int a, b ;
  cout << "Enter a and b:";
  cin >> a >> b ;
  cout << "max=" << process(a, b, max) <<'\t';       //A
  cout << "min=" << process(a, b, min) <<'\t';       //B
  cout << "sum=" << process(a, b, add) <<endl;       //C
```

```
}
int process(int x, int y, int (*fun)(int, int))        // 函数定义
{
  return fun(x, y);                                    //D
}
int max(int x, int y) { return( x>y ? x :y ); }
int min(int x, int y) { return( x<y ? x :y ); }
int add(int x, int y) { return( x + y ); }
```

注意程序的A、B、C三行，三次调用分别传递了三个不同的函数入口地址。利用函数指针调用不同的函数，在调用格式上是通用的（D行），但调用函数却具有灵活性。如在D行，通过指针fun可能调用的是max或min或add函数。

运行时：输入 4 8<Enter>，输出 max=8 min=4 sum=12。

**例9.35** 编写一个通用的积分函数求下列三个定积分的近似值。

$$\int_0^1(\sin(x)+1)\mathrm{d}x \quad \int_0^2(1+x+x^2+x^3)\mathrm{d}x \quad \int_1^{2.5}\frac{x}{1+x^2}\mathrm{d}x$$

用梯形法求定积分的通用公式为：

$$s=h\left(\frac{f(a)+f(b)}{2}+\sum_{i=1}^{n-1}f(a+i\times h)\right), \qquad h=\left|\frac{a-b}{n}\right|$$

其中，$a$和$b$是积分的上限和下限，$n$为积分的分隔数，$h$为积分的步长，$f(x)$为被积函数。通用的积分函数integral( )需要四个参数，它们是$a$、$b$、$n$和$f$函数的指针。

```
#include <iostream.h>
#include <math.h>
double f1(double x)
{  return(sin(x)+1);  }
double f2(double x)
{  return(1 + x + x*x + x*x*x);  }
double f3(double x)
{  return x/(1+x*x);  }
double integral(double(*f)(double), double a, double b, int n )//通用的求积分函数
{ double s, h;
  int i;
  h=(b-a)/n;
  s=(f(a)+f(b))/2;
  for(i=1; i<n; i++) s+=f(a+i*h);
  return(s*h);
 }
void main( )
{ cout<<"f1积分值:"<<integral(f1, 0, 1, 3000)<<endl;
  cout<<"f2积分值:"<<integral(f2, 0, 2, 1000)<<endl;
  cout<<"f3积分值:"<<integral(f3, 1, 2.5, 2000)<<endl;
}
```

程序运行结果：

```
f1积分值:1.4597
f2积分值:10.6667
f3积分值:0.643927
```

从本例可以体会到，使用函数指针可以编写一些通用的函数，使源程序代码简化。

### 9.6.2 返回指针值的函数（指针函数）

函数的返回值可以是整型、实型、字符型量，学习了指针以后，我们知道指针也是一种数据类型，那么函数的返回值当然可以是指针。定义返回指针值的函数格式如下：

```
<类型说明符> * <函数名>( [<参数表>] ) { <函数体> }
```

例如：定义一个返回值为整型指针的函数，格式如下：

```
int *func(...)
{...}
```

函数名之前为返回值类型，本例返回值类型是 int *。可以类推，定义返回float *、char * 类型的函数。注意，如果写成如下形式：

```
int (*func)(...);
```

则表示定义了一个函数指针变量func，它应指向一个返回整型值的函数。

**例9.36**　函数char *find(char *str, char ch)在一个字符串str中找出字符ch第一次出现时的地址，并返回该地址值。

```
#include  <iostream.h>
char *find(char *str, char ch)                    // 返回值类型为 char *
{
  while(*str!='\0')
        if(*str==ch) return(str);
        else  str++;
  return(NULL);                                   // 若找不到，返回空指针
}
void main( )
{
  char  s[ ]="warrior";
  char  *p;
  p=find(s, 'r');  if(p) cout << p<< endl;        // 输出 rrior
  p=find(s, 'i');  if(p) cout << p<< endl;        // 输出 ior
  p=find(s, 'b');  if(p) cout << p<< endl;        // 不输出
}
```

在函数find中，初始时str是字符串的首地址，通过str扫描字符串，如果它指向的字符是待查找字符，则返回该指针；否则指针加1，指向下一字符。如果字符串中不出现字符c，则返回空指针NULL，其值为0。注意主函数中的输出语句cout << p<< endl;，前面我们讲过，一个字符串的本质是从起始地址开始到'\0'结束，如果输出一个字符指针，就是将该指针看成字符串的首指针，然后输出它指向的字符串。

## 9.7 指针小结

学习指针首先必须理解指针的意义，另外对指针的各种定义形式也要熟记。下面以整型量为例，我们将前面学过的各种指针的定义形式和意义做一个简单的总结：

- `int *p;`　一般指针，可指向简单的整型变量，也可指向一维整型数组中的一个元素（因为，数组的每个元素都是一个整型量），与一维数组名等价。
- `int (*p)[M];`　指向含有 M 个元素的一维数组。可指向每行含有M个元素的二维数组的行，即二维数组的行指针，可与二维数组名等价。
- `int * p[M];`　指针数组，有M个元素，每个元素都是整型指针。
- `int **p;`　指向整型指针的指针。
- `int (*p)(int, int);`　p是一个函数指针，可指向参数是两个整型值并且返回整型值的函数。
- `int *p( ){...}`　p为返回整型指针值的函数。

## 9.8 引用和其他类型的指针

### 9.8.1 引用类型变量的说明及使用

引用是C++提供的一种数据类型。定义引用类型变量，实际上是给已有的变量起一个别名，系统不给引用类型的变量分配存储空间，而是与其相关联的变量使用同一空间。定义引用类型变量的一般格式为:

```
<类型标识符> & <引用变量名> = <变量名>;
```

其中，<变量名>为已定义的变量。例如:

```
int count ;  int &refcount = count ;
```

这里refcount是引用类型的变量，是count的别名。refcount和count使用同一存储空间，所以对refcount的任何修改，也同时修改了count的值。

例:
```
int a;
int &ref=a;
a=8;   cout << a <<'\t'<< ref << endl;
ref=10; cout << a <<'\t'<< ref << endl;
```

程序运行后输出:

```
8   8
10  10
```

除了基本类型的变量可以定义引用外，指针类型的变量也可以定义引用，如下例所示。

**例9.37** 为指针类型变量定义引用。

```
#include  <iostream.h>
void  main( )
{
   int a;
   int *p=&a;
   int * &p1=p;  //为指针定义引用，于是p与p1等价
   *p1=5;
   *p = *p+3;
   cout<<a<<','<<*p<<','<<*p1<<endl;
}
```

程序运行后，输出:

```
8, 8 , 8
```

使用引用类型变量的注意事项:

(1) 可以为一个变量定义多个引用:

```
int a;  int &ref1 = a, &ref2 = a ;        // a、ref1和ref2为同一变量
```

(2) 可以定义引用的引用:

```
int  a;  int &ref1=a, &ref2=ref1;         // a、ref1和ref2为同一变量
```

(3) 不能为数组定义引用，但可为数组元素定义引用:

```
int  a[10];
int  &ref=a ;                             // 不正确
int  &ref2=a[5];                          // 正确，ref2与a[5]为同一变量
```

(4) 可为动态申请的空间定义引用（参见9.9节):

```
int  &ref = * new int ;                   // new int返回指针，(* new int) 表示该指针指向
                                             的变量
```

```
ref = 200 ;
cout << ref << endl ;                    // 输出200
delete  &ref ;                           // 释放动态申请的空间
```

### 9.8.2 引用和函数

C++引入引用类型主要是为函数参数传递时提供方便。

#### 1. 基本类型变量和指针变量做函数参数

我们回顾一下用基本类型变量和指针变量做函数参数的两个例子:

例9.8　基本类型量做函数参数。

```
#include<iostream.h>
void swap(int x, int y)
{   int t;
    t=x; x=y; y=t;
}
void main()
{  int x=3, y=9;
   swap(x, y);
   cout<<x<<','<<y<<endl;
}
```

程序运行后，输出:

```
3, 9
```

例9.9　指针做函数参数。

```
#include<iostream.h>
void swap(int *px, int*py)
{   int t;
    t=*px; *px=*py; *py=t;
}
 void main()
 { int x=3, y=9,*p1,*p2;
   p1=&x;p2=&y;
   swap(p1, p2);
   cout<<x<<','<<y<<endl;
}
```

程序运行后，输出:

```
9,3
```

用基本类型量做函数参数时，无法通过参数带回多个结果。用指针做函数参数时，可以通过变量指针带回多个运算结果，但是在编程时，书写通过指针间接访问变量的语句比较烦琐，而且较难理解。下面看一看使用引用类型做函数参数有什么优点。

#### 2. 引用类型做函数参数

例9.38　引用类型做函数参数。

```
#include <iostream.h>
void swap(int &x, int &y)     // A
{
   int t;
   t = x ;  x = y ;  y = t ;
}
void main( )
{
   int  x=3, y=9;
   swap(x, y);
   cout<<x<<','<<y<<endl;
}
```

程序运行后，输出:

```
9, 3
```

注意此例与例9.8比较只是在函数形参的定义时，写成了变量引用形式（参数传递时相当于有int &x=x; int &y=y;），执行效果却与例9.8一样，优点是避免了使用指针做参数时的烦琐。可见使用引用类型变量做参数，增强了程序的可读性和编程的方便性。

### 9.8.3 const类型量

以前，如果在程序中需要经常使用某一常量，可以使用预编译命令定义一个符号常量:

```
#define  PI  3.14
```

于是在程序中可以使用符号常量PI。C++在对源程序预编译时处理PI，即将PI代换成3.14。在程序运行过程中没有PI。

但是在程序的运行中，需要这样的量，就是希望这个量既要具有变量的特性（有变量名，有存储空间，有值，且其作用域与一般变量一样），又要具备常量的特性（不允许修改其值），这就是const类型量。

#### 1. 定义const类型量

例如:

```
const  int  MaxLine=1000;
const  float  PI=3.14;
```

在定义const类型量时，必须赋初值。在使用const类型量时，除了不可以给它赋值以外，与使用一般变量一样。

const类型量与符号常量（#define 定义的符号常量）的区别:

(1) 符号常量由预编译器处理，const类型量由编译器处理。

(2) 在编译时符号常量已经不存在了，而const类型变量依然存在。

(3) 符号常量的作用域从定义位置开始，到文件结束；而const类型量的作用域与一般变量一样。

#### 2. 定义const类型指针

有三种不同的方法来定义const型指针，其作用和含义是不同的。

**(1) 指向常量的指针。**

定义格式为: const <类型标识符> * <指针变量名>;

注意，定义时const 的位置在最前面。这种类型指针的意义是，不能通过该指针改变它所指向的内容。例如:

```
const int *p;
const int a=15;
int b;
```

有了上述定义后，下述的操作有些是合法的，有些是不合法的。

```
p = &a;          // 合法，p可以指向常量
p = &b;          // 合法，p也可以指向变量
*p = 18;         // 不合法，不能通过p间接地给它指向的变量赋值
b = *p;          // 合法，可以取出p指向的值
b = 18;          // 合法，b是被p指向的变量，b是独立的，可以给它赋值
```

通过上面的说明，可知定义指向常量的指针，只限制指针的间接赋值操作，仅仅不允许做“*p=...”操作，即不允许通过指针间接给它指向的变量赋值。而不能限制指针指向的量的自身的操作特性，如b=18; 是合法的。

**(2) 指针常量。**

定义格式为: <类型标识符> * const <指针变量名>;

注意，定义时const 的位置在指针变量的前面。这种类型指针的意义是，不能改变指针的指向，不允许给这种类型的指针重新赋值。例如:

```
int a, b ;
int * const p = &a;   // 必须在定义时给p初始化，让p固定指向一变量
*p = 15;              // 合法，可以通过p间接访问它指向的变量a
```

```
b = *p;                    // 合法，可以取出p指向的值
p = &b;                    // 不合法，不可以给指针p重新赋值
a = 8;                     // 合法，a是被p指向的变量，a是独立的，可以直接访问变量a
```

定义指针常量，同时必须给该指针赋初值，不能改变指针的指向，即仅仅不允许做“p=...”操作。

**(3) 指向常量的指针常量。**

定义格式为：const <类型标识符> * const<指针变量名>;

注意在格式中有两个const。这种类型指针的意义是，不能通过该指针改变它所指向的内容，也不能改变指针的指向。例如:

```
int a, b ;
const int * const p = &a;  // 定义时，必须给它赋初值，即让它有一个指向
*p = 15;                   // 不合法，不可以给指针 p 指向的内容赋值
b = *p;                    // 合法，可以取出p指向的值
p = &b;                    // 不合法，不可以给指针 p 赋值
a = 8;                     // 合法，a是被p指向的变量，a是独立的，其自身的操作特性不受影响
```

若p为指向常量的指针常量，则“p=...”和“*p=...”操作都是非法的。

### 3. 小结

若有int a, b ; 则

- 指向常量的指针如下所示:

```
const int *p;
*p=...                     // 不合法
```

- 指针常量如下所示:

```
int * const p = &a;        // p是指针常量，固定指向a
p=...                      // 不合法
```

- 指向常量的指针常量如下所示:

```
const int * const p = &a;  // 注意有两个 const
即 p=...                   // 不合法
   *p=...                  // 不合法
```

**例9.39** 使用指向常量的指针的意义。

```
#include  <iostream.h>
void my_strcpy(char * dest, const char *source)  // A 注意第二个参数的定义
{
   while( *dest++ = *source ++);
}
void main( )
{
   char a[20] = "How are you!";
   char b[20];
   my_strcpy(b, a);
   cout << b << endl;
}
```

定义const类型指针的主要目的是提高程序的安全性。上述程序中函数void my_strcpy(char * dest, const char *source)的功能是，将第二个参数指针指向的字符串赋给第一个指针指向的字符串。因此源字符串source的内容应该受到保护（即不允许被改变），所以将它定义成指向常量的指针，即不允许做*source=...的操作，这样就可以达到保护源字符串不被改变的目的。如果误写了 *source = ...，则编译时会报错。这时，就不需要人工来保证操作的正确性了。

## 9.9 存储空间的动态分配和释放

在本节内容之前，变量存储空间的分配及释放是由系统自动完成的，不需用户干预。对于静态变量，编译时确定其空间，在程序开始运行时分配其空间，程序结束时释放空间。对于动态变量，程序在运行时由系统动态分配、撤销其空间。如函数内部的局部动态变量，进入函数时，系统自动分配空间，函数执行结束时，系统释放其空间。不需编程者干预。

我们可以通过变量名来访问变量，称为变量的直接访问；也可以通过指针来访问变量，称为变量的间接访问。

现在我们学习一种在程序运行过程中，动态申请变量存储空间的方法。程序动态申请空间是由编程者在编程时安排的。申请多少空间由运行时的具体情况而定。这时，只能通过指针间接访问变量了。动态申请的存储空间在程序结束前的适当时刻，必须由编程者安排释放其空间。

以前有同学想根据问题的实际规模来定义数组的大小，试图这样做：

```
int n;
cin >> n ;
int a[n];
```

这是不可以的。因为C++编译时必须确定数组空间的大小，但此段程序在运行时，才能输入变量n的值，也就是说在编译时是不知道n的大小的，因此在编译时无法确定数组空间的大小。那么能否根据问题的实际规模来确定实际需要使用的存储空间的大小呢？回答是肯定的，学习了变量空间的动态分配和释放，我们就可以解决这个问题了。

### 9.9.1 new和delete运算符

下面学习new和delete运算符的用法。new用于动态申请存储空间。delete用于释放由new申请的存储空间。

#### 1. new 运算符

new运算符常用的四种格式及意义：

格式一：<指针变量> = new <类型名>;

意义：申请一个<类型名>类型变量的空间，返回该空间的起始地址，并将该地址赋给<指针变量>。

例：
```
int *p;
    p = new int ;
    *p = 8 ;
    cout << *p ;
```
该空间只能通过指针p间接访问

格式二：<指针变量>= new<类型名> (<值>);

意义：申请一个<类型名>类型变量的空间，以<值>初始化该空间，返回该空间的起始地址，并赋给<指针变量>。与格式一的区别在于给定了变量的初值。

例：
```
int  *p, i ;
    p = new int(8) ;          该空间中的初值为 8
    i = *p;                   该空间只能通过指针p间接访问
```

格式三：<指针变量> = new <类型名> [<表达式>];

意义：申请<表达式>个<类型名>类型变量的空间，返回该空间的起始地址，并赋给<指针变量>，相当于分配<类型名>类型的一维数组空间。

例：
```
int  *p;
    p =  new int[10] ;              申请了10个整数（即一维数组）的空间
    ......  p[i] ..... ;
    ......  *(p+i) ..... ;         可以将该空间看成一维数组来访问
```

格式四：<指针变量> = new <类型名> [<表达式1>] [<表达式2>];

意义：申请<表达式1>×<表达式2>个<类型名>类型变量的空间，并返回该空间的起始地址（返回行指针），赋给<指针变量>，相当于分配二维数组空间。

```
例: int  (*p)[4] ;
    p = new int[3][4];                    申请了二维数组的空间，返回二维数组指针
    ......  p[i][j] ..... ;
    ......  *(*(p+i)+j) ..... ;          可以将该空间看成二维数组来访问
```

由new动态申请的存储空间，在程序结束前必须通过 delete 释放。

### 2. delete 运算符

delete 运算符的常用的两种格式:

格式一: `delete <指针变量>;`

意义：释放一个基本类型变量的空间。用于释放由上述new运算符格式一及格式二分配的空间。

```
例: int *p1, *p2;
    p1 = new int ;  p2 = new int(9) ;
    *p1 = 8;
    delete p1 ;  delete p2 ;
```

格式二:  `delete [N ] <指针变量>;`　　// N可省略

意义：释放一个指针指向的数组空间，该数组有N个元素。

```
例: int  *p;                              // 定义指针
    p = new int[10] ;                     // 分配数组空间
    ...  p[i] ... *(p+i)... ;             // 使用数组空间
    delete [ ] p ;                        // 释放数组空间
例: int  (*p)[4] ;
    p = new int[3][4] ;
    delete [3] p ;
```

**例9.40** 使用动态数组空间。

```
#include  <iostream.h>
void  main( )
{
   int n, *p, i;
   cin >> n ;
   p = new int[n] ;                  // 申请数组空间
   for(i=0; i<n; i++)                // 使用数组空间
       cin >> p[i];                  // 或改写为 cin >> *(p+i);
   for(i=0; i<n; i++)
       cout << p[i] << '\t';         // 或改写为 cout << *(p+i);
   cout << '\n';
   delete [ ] p ;                    // 释放数组空间
}
```

## 9.9.2 使用new和delete运算符的注意事项

应注意以下几点:

(1) 使用new运算符格式二在申请空间的同时初始化该空间中的值。使用new运算符格式一、格式三和格式四，分配的存储空间中的值是不确定的，申请空间完毕，必须用赋值语句通过指针间接地对分配的空间赋值。例如:

```
int *p1=new int(5), *p2=new int ; *p2 = 8 ;
```

（2）使用new运算符动态申请空间，不是每次都成功的，也就是说不一定每次都能申请到。所以，一般编写正规程序时，为保证程序执行正确，每次使用new申请空间后，都要测试是否成功。如果成功，new返回被申请到的空间的首指针；如果失败，则返回空指针NULL，即0。此时可以做相关处理，或终止程序执行，或进行出错处理。

```
float *p;
p = new float[10] ;
if( p == NULL )  // 括号中的条件可写成 p= =0
{
     cout << "动态分配不成功，终止执行! \n";
     exit(3);
 }
 ......
delete [ ] p ;
```

## 9.10 链表及其应用

### 9.10.1 结构体及指针

#### 1. 结构体类型变量指针

前面我们介绍过结构体类型，一个定义结构体类型的例子如下：

```
// 类型的定义放在文件student_type.h 中
struct  student
{
     int   num;              // 学号
     char  name[20];         // 姓名
     int   age;              // 年龄
     char  sex;              // 性别
     int   score;            // 成绩
};
```

类型定义完毕，就可以定义该类型的结构体变量了，如：`struct student stud1, stud2;`定义了两个结构体变量stud1和stud2。此定义可简化为：`student stud1, stud2;`，即缺省struct。

我们知道，可以定义简单变量的指针，同理，也可以定义结构体变量的指针，定义格式为：

<结构体类型标识符> * <结构体变量名>;

如：`student *p1, *p2;`。那么如果想让p1和p2分别指向stud1和stud2，则应该做如下赋值：`p1=&stud1; p2=&stud2;`。此时 *p1与stud1等价，*p2与stud2等价。与基本类型变量的指针做类比就可以理解，如：若有`int x, *p ; p = &x;`，则 *p 与 x等价。

可以通过结构体变量名访问成员，也可以通过结构体指针访问它所指向的结构体变量的成员，比较下述三种方法，参见图9-24。

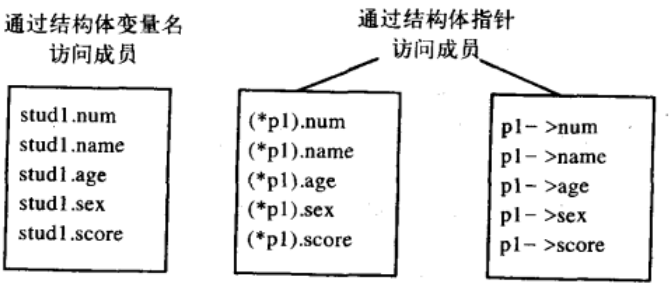



图9-24 结构体变量成员的三种访问方法

在第三种方法中，p1指向结构体变量stud1，通过运算符“->”（指向结构体成员运算符），访问结构体变量stud1的各个成员值。

### 2. 指向结构体数组元素的指针

与整型数组类比，可以定义结构体数组，并且定义一个指针，让它指向结构体数组的元素，请看下面的例子。

**例9.41** 定义结构体数组并初始化，用一指针指向其元素，循环输出全体数据。

```
#include  <iostream.h>
#include  "student_type.h"          // 如上
void main( )
{  student studs[3]=                // 定义结构体数组并初始化
   {     {10101, "Li Nin",   18, 'M', 88},
         {10102, "Zhang Fun",19, 'M', 99},
         {10104, "Wang Min", 18, 'M', 70},
   };
   student *p;                      // 定义结构体指针
   p = studs;                       // A
   for(int i=0; i<3; i++, p++)      // B
          cout << p->num <<','
               << p->name<<','
               << p->age <<','
               << p->sex <<','
               << p->score << endl ;
}
```

结构体数组及其指针的含义如图9-25所示。程序中的A行功能是令p指向结构体数组第0个元素，B行p++的含义是令p指向下一结构体元素。

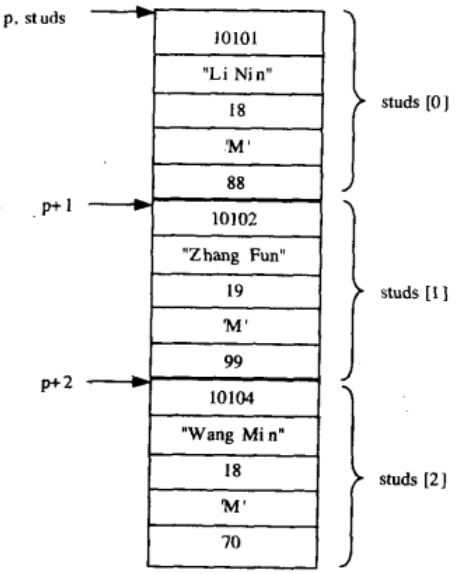



图9-25　结构体类型指针的意义

## 9.10.2　链表概念的引入

在处理一些实际问题时，如编写一个通用的处理一个班学生数据的程序，一般使用结构体数组，由于每个班学生数不同，在考虑通用性的前提下，应需预留足够大的数组空间。如每班人数可能是20、30、45、50人，则定义数组为50个元素，这样可以用此数组处理不同人数的班级。但与此同时带来的问题是处理小班时，多余的元素占用的空间浪费。这个问题可用动态申请数组空间解决。也就是说，在编程时可以根据每班的实际人数动态申请数组空间。但是数组占用连续的存储空间，有时未必能申请到过大的连续的存储空间，这时的解决办法是使用链表结构。链表结构犹如现实生活中的链子，由若干环节组成，每个环节称为一个结点（或称为节点）。在每个结点中存放一个学生的数据，每个结点内部的空间是连续的，但一个结点与另一个结点的存储空间不一定要连续，结点之间可以用指针连接起来。

对于某些问题的算法，使用链表结构，较之使用数组结构更有优势。还是以上例说明，假如在数组结构中，学生数据是按学号顺序排列的，现在欲添加一个学生数据，要求添加数据后的数组仍然按学号顺序排列，若这个学生的学号正好是已有学号的中间值，那么就要将数组后半部分的数据依次向后挪动，这是费时费力的。使用链表结构就不需要挪动数据。但是使用链表也有一些不足之处，如耗用多余的构成链表的指针空间，另外，对有些问题使用链表结构，对写算法不利。在解决实际问题时，要靠长期编

程积累起来的经验来决定使用哪一种数据结构较为有利。

在C++语言中用结构体实现链表环节，也就是说每个结点为一个结构体变量。当然，为了形成链表，还要增加一个指向下一结点的指针成员。链表结构示意图如图9-26所示。

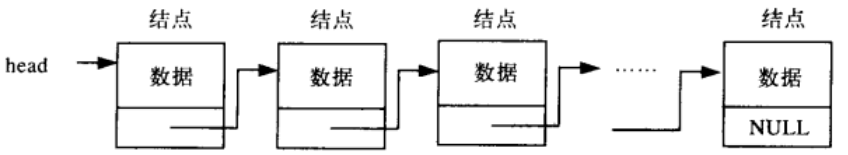



图9-26 单向链表示意图

在链表结构中，有一个首指针head指向链表首结点，每一个结点中有一个指向下一结点的指针，最后一个结点的下一结点指针值为NULL，即空指针，表示链表到此结束。

例如，定义链表结构结点的结构体类型如下，注意画线部分的定义：

```
struct student
{
     int       num;
     char      name[20];
     int       age;
     char      sex;
     int      score;
     student  *next;              // 指向下一结点的指针
};
```

### 9.10.3 有关链表的算法

对链表的常规操作有创建无序/有序链表、遍历链表、查找结点、插入结点、删除结点、释放链表空间等。在后面的程序中，我们用以下的通用结点结构，该结构的定义放在头文件node_def.h中，文件内容如下：

```
struct node
{    int data;      // 数据
     node *next;    // 指向下一结点的指针
};
```

下面我们一一介绍有关链表的常用算法。

#### 1. 创建无序链表

创建链表就是从无到有，动态申请结点空间，加入链表，最终建立一个链表的过程。下面我们“硬性”地建立一个链表，体会一下链表的概念。

**例9.42** 动态申请，建立链表，输出各结点数据，释放各结点空间。

```
#include  <iostream.h>
#include  "node_def.h"
void main( )
{  node  *head, *p1, *p2;
   head = new node;
   p1 = new node;
   p2 = new node;
   head->data = 1000;  head->next = p1;        // 建立第1个结点
   p1->data = 1001;    p1->next = p2;          // 建立第2个结点
   p2->data = 1002;    p2->next = NULL;        // 建立第3个结点
```

```
    while(head!=NULL)                                    //A
    {     cout << head->data << endl;
          p1=head;
          head = head->next;
          delete p1;
    }
}
```

程序在A行之前，建立的链表如图9-27所示。

程序运行后，输出结果如下：

```
1000
1001
1002
```

在实际应用中，往往是动态建立链表，下面的例子是循环输入数据，若该数据不为－1，就动态申请结点，以该数据为结点数据，动态地向链表的尾部追加结点，当数据输入结束时（即输入－1时），链表的创建过程结束。加入结点时，分三种情况：（1）首结点的建立；（2）中间结点的插入；（3）尾结点的处理，链表建立结束时，将尾结点的next成员赋值为空指针NULL。在程序中用到三个指针，它们的意义是：head 指向链表首结点，p2 指向建立过程中的链表尾结点，p1 指向新开辟的结点（见图9-28）。

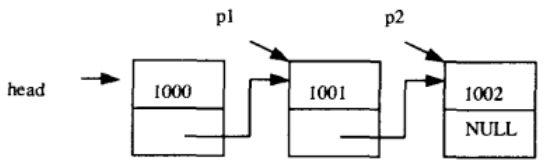



图9-27　已建链表

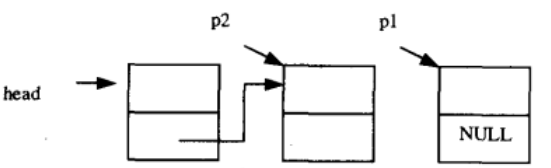



图9-28　创建链表过程示意

请看函数Create( )，并注意各部分的说明，此函数在头文件create.h中：

```
node *Create( )                                  // 返回值是：链表的首指针
{     node *p1, *p2, *head;
      int a;
      head = NULL;
      cout <<"产生一条无序链表，请输入数据，以－1 结束:";
      cin >> a ;                                 //输入第1个数据
      while( a != -1 )                           //循环输入数据，建立链表
      {     p1 = new node;
            p1->data = a;
            if(head==NULL)                       //（1）建立首结点
            { head=p2=p1; }
            else                                 //（2）处理中间结点
            { p2->next=p1; p2=p1; }
            cin >> a;                            //输入下一个数据
      }
      if(head != NULL)
      p2->next=NULL;                             //（3）处理尾结点（如果第1次就输入－1，则此句不做）
      return(head);                              //返回创建链表的首指针
}
```

### 2. 遍历链表（查找结点）

遍历链表就是依次访问链表的各个结点。查找结点就是顺序查找某结点的值，如果存在则返回该结

点的指针，否则返回空指针。在头文件print_search.h中，列出了两个函数，它们是输出链表各结点数据函数Print( )以及在链表中查找某值x的函数Search( )，请读者参照程序中的注释阅读这两个函数。

```
void Print( const node *head )                  // 不允许通过head修改链表中结点的值
{  const node *p;                               // 不允许通过p修改它指向的结点的值
   p=head;
   cout << "输出链表中各结点数据: ";
   while( p!=NULL )
   {       cout << (p->data) << '\t';
           p=p->next;
   }
   cout << endl;
}
const node * Search( const node *head, int x )  // 不允许通过head修改链表中结点的值
{ const node *p;                                // 不允许通过p修改它指向的结点的值
  p=head;
  while( p!=NULL )
  {       if(p->data == x) return p;            // 若找到，则返回该结点指针
          p = p->next;
  }
  return NULL;                                  // 若找不到，则返回空指针
}
```

### 3. 删除结点

删除结点就是删除链表中满足一定条件的结点，例如删除结点的data值等于num的结点（为简单起见，我们假定只删除第一个值等于num的结点）。请看函数，注意各部分的说明，此函数在头文件delete.h中。注意在删除结点时，也分三种情况：（1）链表为空链表；（2）待删除的结点是链表的首结点；（3）删除其他结点。如图9-29所示，找到待删除的结点（由p2指向），然后删除它。

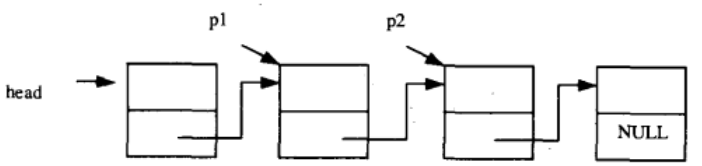



图9-29 删除结点过程示意

```
node *Delete_one_node( node *head,  int num )
{
     node *p1, *p2;
     if( head == NULL)                          // 处理链表为空时的状况
     {      cout << "链表为空、无结点可删!\n";
            return(NULL);
     }
     if( head->data == num)                     // 若待删除的结点为首结点
     {      p1=head;
            head=head->next;
            delete p1;
            cout << "删除了一个结点!\n";
     }
     else                                       // 删除其他结点
```

```
    {     p1=head;
          p2= head->next;                    // p1指向的结点在p2指向的结点之前
          while(p2->data != num&&p2->next !=NULL )  //先循环查找待删除结点
          {                                  // 若p2指向的结点不是待删除结点，并且也不是尾
                                                结点时循环
                p1=p2;
                p2=p2->next;                 // p1、p2均向后挪动一个结点
          }
          if(p2->data == num)                // 若找到，则删除p2指向的结点
          {                                  // 注意：若p2指向的结点是链尾结点，这里的处理
                                                同样适用
                p1->next = p2->next;
                delete p2;
                cout << "删除了一个结点!\n";
          }
          else cout<<num<<"链表上没有找到要删除的结点!\n";
     }
    return(head);
}
```

### 4. 释放链表

释放链表就是释放链表全体结点的空间。请看函数Delete_chain( )，并注意各部分的说明，此函数放在头文件deletechain.h中。

```
void  Delete_chain( node *head )
{  node *p1;
   while( head)
   {      p1=head;
          head=head->next;
          delete p1;
   }
}
```

### 5. 插入结点

为了建立有序链表，我们在这里介绍一个插入结点的函数Insert( )，此函数执行的前提是，假定已建立了一个结点值为升序的链表，调用此函数插入一个新结点后，链表的值仍然保持升序。插入结点时，同样分三种情况：（1）原链表为空链表；（2）插入在首结点之前；（3）插入在链表中间或尾部。请看函数，并注意各部分的说明，此函数放在头文件insert.h中。如图9-30所示，将p指向的结点插入到p1和p2之间。

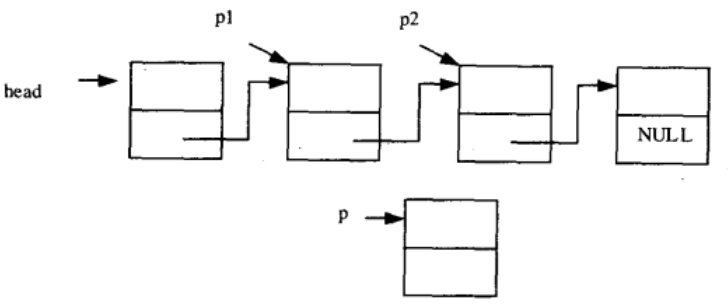



图9-30　插入结点过程示意

```
node *Insert(node *head,  node *p)
{                                                    // 把结点p插入到链表head中，使链表仍
                                                        然保持升序
  node *p1, *p2;
  if(head==NULL)                                     // 如果插入前（原始链表）为空链表
  {     head=p;
        p->next=NULL;
        return(head);
  }
  if( head->data >= p->data )                        // 插在链表首部
  {     p->next = head;
        head = p;
        return(head);
  }
  p2=p1=head;                                        // 插在链表中间或尾部
  while( p2->next && (p2->data) < (p->data ) )       // 查找待插入位置
  {     p1=p2; p2 = p2->next;  }
  if( (p2->data) < (p->data) )                       // 插在链表尾部
  {     p2->next = p;
        p->next = NULL;
  }
  else                                               // 插在链表中间，p2之前
  {     p->next = p2;
        p1->next = p;
   }
   return(head);
}
```

### 6. 创建有序链表

创建有序链表就是循环输入数值，调用插入结点函数Insert( )，将新建结点插入到链表中，使链表保持升序。请看函数，并注意各部分的说明，此函数在头文件create_sort.h中。

```
node *Create_sort(void)
{
  node *p1, *head=NULL;
  int a;
  cout <<"产生一条有序链表，请输入数据，以-1结束：";
  cin >> a;
  while( a!=-1 )
  {     p1=new node;
        p1->data = a;
        head=Insert(head, p1);
        cin >> a;
  }
  return(head);
}
```

### 7. 在主函数中，测试上述函数

下面的主函数用于测试上述各函数。

**例9.43** 链表各算法测试函数。

```
// Li0943.cpp
#include  <iostream.h>
```

```
#include  "node_def.h"
#include  "create.h"
#include  "print_search.h"
#include  "delete.h"
#include  "insert.h"
#include  "deletechain.h"
#include  "create_sort.h"
void main( )
{  node *head;
   int num;
   head=Create( );                          // 建立一条无序链表
   Print(head);                             // 输出链表
   cout << "输入待删除结点上的整数: \n";
   cin >> num;
   head=Delete_one_node(head, num);
   Print(head);                             // 删除一个结点后，输出链表
   cout << "输入要查找的整数: \n";
   cin >> num;
   if(Search(head, num)!=NULL)
         cout<<num<<" 在链表中\n";
   else
         cout<<num<<" 不在链表中\n";
   Delete_chain (head);                     // 释放上述链表空间
   head=Create_sort( );                     // 建立一条有序链表
   Print(head);                             // 输出链表
   Delete_chain (head);                     // 释放有序链表空间
}
```

对于本例，我们将各函数放在头文件中。当然，在编程时，可以将上述各函数均放在一个cpp源文件中。

## 习题9

本章习题要求用指针的方法处理（如通过指针访问数组元素、将数组或变量的起始地址传递到被调函数），并且一般要求在主函数中输入输出数据，在被调函数中完成处理工作。

1. 在主函数中输入任意三个整数存入变量a、b、c中，调用函数swap(int *, int *, int *)实现将三个变量值进行交换（要求函数的参数为三个变量a、b、c的地址），达到从小到大排序的目的，即变量a中的值是初始三个值中的最小者，变量c中的值是初始三个值中的最大者，最后输出a、b、c三个变量的值。
2. 在主函数中输入一个字符串str，调用函数统计字符串中出现的字母（含大小写）、数字、空格以及其他字符出现的次数，在主函数中输出统计结果。要求写两个版本的程序：（1）用指针做参数带回统计结果，（2）用引用做参数带回统计结果。
3. 在主函数中输入$n$个整数到数组中，调用函数（要求被调函数参数为：（1）数组名，（2）数组元素的个数）完成将数组中的最小值与第0个元素对调，将数组中的最大值与最后一个元素对调，主函数中输出调换后的数组。
4. 在主函数中输入10个整数到数组中，调用函数完成将数组元素循环移动k位（要求函数参数为：（1）数组名，（2）数组元素个数，（3）循环移动的位数$k$）。当k>0时，实现循环右移；当$k$<0时，实现循环左移。循环右移一位的意义是：将数组全体元素向后一个下标位置移动一个元素的位置，原数组最后一个元素移动到数组最前面第0个元素位置。提示：当$k$<0时，转换成等价的循环右移。
5. 在主函数中输入一个字符串str1，调用函数将str1中的下标为奇数的字符取出构成一个新的字符串放入字符串str2中（要求被调函数参数为str1和str2），在主函数中输出结果字符串str2。

6. 在主函数中输入一个字符串main_str，调用函数将main_str中最长的单词取出放入字符串sub_str中（要求被调函数参数为main_str和sub_str），在主函数中输出结果字符串sub_str。假定输入的字符串main_str中的单词以一个或多个空格隔开。例如：若输入字符串“She is a nice girl.”，则结果输出字符串“girl.”。
7. 在主函数中定义并初始化一个4×5的二维数组，调用函数求出该二维数组的鞍点，并通过行列下标变量指针返回鞍点的行列下标值。所谓鞍点是指二维数组中的某个元素，该元素在它所在的行中最大，在它所在的列中最小。要求被调函数参数为：（1）二维数组名，（2）鞍点的行下标变量的指针，（3）鞍点的列下标变量的指针。在主函数中定义行、列下标变量，用于存放鞍点的行、列下标值。
8. 将第7题被调函数参数变化为：（1）二维数组名，（2）鞍点的行下标变量的引用，（3）鞍点的列下标变量的引用，完成相同的功能。
9. 编写一个函数palin( )用来检查一个字符串是否是正向拼写与反向拼写都一样的“回文”(palindromia)。如“MADAM”就是一个回文。若放宽要求，即忽略大小写字母的区别、忽略空格及标点符号等，则像“Madam,I’ m Adam”之类的短语也可视为回文。编程要求：（1）在主函数中输入字符串。（2）将字符串首指针作为函数参数传递到函数palin( )中。当字符串是回文时，要求函数palin( )返回“真”值，否则返回“假”值。（3）若是回文，则在主函数中输出yes。若不是回文，则在主函数中输出no。
   提示：在函数palin( )中，定义两个指针变量，分别指向字符串首部及尾部，判断它们指向的字符相等后，首指针head向后移动一个字符位置，尾指针tail向前移动一个字符位置，遇空格或标点符号等则跳过。直到能判断出结果。
10. 编写一函数实现两个字符串s1、s2的比较，即自己编写一个函数my_strcmp(s1, s2)，如果s1=s2，则函数返回值为0；如果s1≠s2，则返回第一个不同字符的ASCII码差值。例：“BOY”与“BOSS”，返回6。（因‘Y’ - ‘S’ == 6 )。编程要求：在主函数中输入两个字符串s1、s2，调用my_strcmp( )函数，并且my_strcmp(s1, s2)的返回值也在主函数中输出。
11. 任意输入三个字符串，按从小到大的顺序输出。编程要求：（1）在主函数中定义三个字符数组str1[ ]、str2[ ]、str3[ ]，输入三个字符串分别存入这三个字符数组中。（2）将str1、str2、str3三个字符数组的指针传递给被调函数。（3）在被调函数swap(char *, char *, char *)中做字符串变量值的交换。（4）返回主函数后，变量str1中是初始三个字符串中的最小者，变量str3中是初始三个字符串中的最大者，输出str1、str2、str3三个字符串的值。
12. 在主函数中定义一个含有12个指针的指针数组，令它的数组元素分别指向12个由月份组成的字符串常量，如第0个元素指向“January”。若输入数字月份，则输出相应的月份字符串。如输入“8”，则输出“August”。
13. 在main( )函数中，首先调用函数create( )建立一个无序链表，结点仅包含一个整型数据和指向下一结点的指针，循环输入整数，依次加入到链表中。调用函数reverse( )将该链表转换成逆序链表，将原链表头变成新链表尾，即将原链表尾变成新链表头。调用函数print( )输出新链表中各结点的值。最后调用free( )释放链表空间。要求自行设计三个函数的参数和返回值（均为指针）。

# 第二部分　面向对象的程序设计

## 第10章　类和对象

### 10.1　类和对象的定义

#### 10.1.1　从结构体到类

现实世界中，任何事物都可以称为对象。比如：人、汽车、几何图形等。每一个对象都具有静态属性和动态属性，比如“人”这个对象，具有姓名、性别、年龄等静态属性，又具有一些动态行为，如学习、走路、吃饭等动作。

在第8章，我们把描述对象的静态属性抽象为结构体，如：

```
struct SPerson                    // 定义结构体类型
{
        char name[20];            // 姓名
        char sex;                 // 性别        } 三个数据成员
        int  age;                 // 年龄
};
```

结构体是一种导出的抽象数据类型，如果要描述一个具体的人，还要定义具体的结构体变量，我们称之为对象，将人的具体属性值赋给对象的相应成员。当然也可以定义结构体变量的指针：

```
struct SPerson  a, b, *pa, *pb;        // 定义结构体变量、指针
pa = &a;                               // 令pa指向a
pb = &b;                               // 令pb指向b
```

通过对象a和指针pa访问结构体成员，可用下述三种方法之一：

```
a.name      pa->name      (*pa).name
a.sex       pa->sex       (*pa).sex
a.age       pa->age       (*pa).age
```

如果结构体变量a代表一个学生，学生的姓名为"wang"，性别为'F'，年龄为20岁，则可任选下述方法给a赋值：

```
(1) strcpy(a.name, "wang");  a.sex='F';  a.age=20;
(2) strcpy(pa->name, "wang");  pa->sex='F';  pa->age=20;
(3) strcpy((*pa).name, "wang");  (*pa).sex='F';  (*pa).age=20;
```

现在我们要使用面向对象的方法编程，把对对象可进行的操作也加入到对象的描述中，这样就抽象出了“类”。类是面向对象程序设计的基础。

#### 10.1.2　类和对象的定义形式

类的定义形式如下：

```
class <类名>
{
```

```
        [[private:]
              <数据成员及成员函数>]
        [protected:
              <数据成员及成员函数>]
        [public:
              <数据成员及成员函数>]
};
```

其中，class是定义类的关键字，<类名>是类的标识符，实际上它是新的数据类型的类型名。private、protected和public表示类的成员的访问权限，分别是私有的、保护的和公有的。三种访问权限的类成员，在类的定义时出现的先后次序可以随意、可以任选，而且可以出现多次，如一个类的定义为：public权限成员段、private权限成员段、public权限成员段。注意：数据成员和成员函数的缺省访问权限是私有的。我们可以建立头文件person.h，并在其中定义一个描述“人”的类：

**例10.1**　定义Person类，源程序文件名为person.h。

```
#include <iostream.h>
#include <string.h>
class  Person
{
private:                                              //此处，private可缺省
          char Name[20];                              //姓名   ┐
          char Sex;                                   //性别   ├ 三个数据成员
          int  Age;                                   //年龄   ┘
public:
          void SetData(char na[ ], char s, int a)     //以下定义了四个成员函数
          {
                strcpy(Name, na);                     //直接访问Name
                Sex=s;                                //直接访问Sex
                Age=a;                                //直接访问Age
          }
          void GetName( char *na )
          {    strcpy(na, Name);      }
          char GetSex( )
          {    return Sex;            }
          int GetAge( )
          {    return Age;            }
};                                                    //注意：类定义结束处必须有分号
```

在Person类中定义了三个私有数据成员和四个公有成员函数，公有成员函数的作用是对私有数据成员进行存取。类名Person是一种新的数据类型标识符，与结构体类似，我们可以定义该类型的对象（即该类的变量）用于描述若干个具体的“人”。定义对象的格式是：

```
<类名>  <对象列表>;
```

如：Person a, b ；定义了a、b两个对象。

当然，也可以定义对象的指针和对象数组，如：

```
Person *pa, *pb, *px, x[10];          //pa和pb是两个Person类型指针，x是对象数组
pa=&a; pb=&b; px=x ;                  //pa和pb指向对象a和b，px 指向对象x[0]
```

### 10.1.3　对象成员的访问

我们注意到，上述三个数据成员的访问权限是private（私有的），C++规定在类内可以直接访问它们，在类外不可直接访问它们。所谓“类内”是指在类的成员函数中；所谓“类外”是指在类的成员函数之

外，通过对象或通过对象指针访问对象的成员。

如在上述Person类的成员函数SetData( )内部可以直接访问成员Name、Sex和Age。

关于private、protected和public三种成员的访问权限的规定如表10-1所示。

表10-1 类成员的访问特性

| 成员的访问权限 | 类内访问特性 | 类外访问特性 |
|---|---|---|
| private | 可直接访问 | 不可直接访问 |
| protected | 可直接访问 | 不可直接访问 |
| public | 可直接访问 | 可直接访问 |

由表10-1可以看出，在类内可以直接访问具有任何访问权限的成员，在类外不可直接访问private和protected两种成员，仅仅可以直接访问public成员。

如a.Name, (*pa).Name, pa->Name 是非法的；而a.SetData( ), (*pa).SetData( ), pa->SetData( )是合法的。如果将数据成员定义成公有的:

```
class  Person
{
public:
        char Name[20]; //姓名
        char Sex;      //性别      } 三个数据成员
        int  Age;      //年龄
......
};
```

则此时a.Name, (*pa).Name, pa->Name就是合法的，即在类外访问公有成员是合法的。

### 10.1.4 成员函数的定义

在面向对象程序设计中，一般将数据成员设计成私有的，以便实现类的数据的“隐藏”。也就是用户在使用“类”时，不必去关心类中具体的数据结构，而只要使用好类的成员函数即可。我们也将类的成员函数称为类的方法。用户有时需要修改私有数据成员的值，那么“类”应提供访问私有数据成员的方法，一般设计成公有成员函数，这就是公有成员函数的作用之一。如上述SetData( )成员函数用于修改类的私有数据成员，而GetName( )、GetSex( )和GetAge( )用于在类外获取私有数据成员的值。我们可以把类看成“黑箱子”，私有成员被“隐藏”在黑箱子中，要想存取私有成员，必须通过黑箱子上的专用窗口即公有函数接口来进行。

类的成员函数的定义可以在类体内，也可以在类体外。在例10.1中，我们看到了在类体内定义成员函数的形式。当然也可以在类体外定义成员函数，如将Person类的定义修改成如下形式:

**例10.2** 定义Person类，在类体外定义成员函数，程序文件名为person.h。

```
#include <iostream.h>
#include <string.h>
class Person
{
          char Name[20];                          //姓名
          char Sex;                               //性别
          int  Age;                               //年龄
public:
          void SetData(char [ ], char , int );    } 在类体内进行成员函数说明
          void GetName(char *);
          char GetSex( );
          int GetAge( );
```

```
};
void Person::SetData(char na[ ], char s, int a) // 在类体外定义成员函数
{    strcpy(Name, na);
     Sex=s;
     Age=a;
}
void Person::GetName( char *na )                  // 在类体外定义成员函数
{    strcpy(na, Name);  }
char Person::GetSex( )                            // 在类体外定义成员函数
{    return Sex;  }
int  Person::GetAge( )                            // 在类体外定义成员函数
{    return Age;   }
```

注意程序中的画线部分，在类体外定义成员函数必须在函数名前加类的限定 Person::，表示函数所属的类，如void Person::GetName( char *n ) 表示函数GetName( )是类Person的成员函数。

类定义结束后，我们就要使用类了。下面，请看一个使用类的例子。

**例10.3** 测试Person类。

```
#include "person.h"                               // 包含例10.1或例10.2中的头文件
void main( )
{
     Person a, *pa;
     char name[20];                               // 以下通过对象访问成员
     a.SetData("Cheng", 'F', 20);
     a.GetName(name);
     cout << "Name: " << name <<endl;
     cout << " Sex: " << a.GetSex( ) << endl;
     cout << " Age: " << a.GetAge( ) << endl;
     pa=&a;                                       // 以下通过指针访问成员
     pa->SetData("Zhang", 'M', 18);
     pa->GetName(name);
     cout << "Name: " << name <<endl;
     cout << " Sex: " << pa->GetSex( ) << endl;
     cout << " Age: " << pa->GetAge( ) << endl;
}
```

**例10.4** 定义并测试长方形类CRect，长方形由左上角坐标(left, top)和右下角坐标(right, bottom)组成。

```
#include <iostream.h>
#include <math.h>
class  CRect                                      // 定义长方形类
{
   private:
       int left, top, right, bottom ;
   public:
       void setcoord(int, int, int, int);
       void getcoord(int *L, int *T, int *R, int *B)  //注意: 形参为指针变量
       { *L = left; *T = top; *R = right; *B = bottom;  }
       void print(void)
       {    cout<<"Area = "<<abs(right-left)*abs(bottom-top)<<endl;  }
};
void CRect::setcoord(int L, int T, int R, int B)
{    left=L; top=T; right=R; bottom=B;  }
```

```
void main(void)
{    CRect r, rr;
     int a, b, c, d ;
     r.setcoord(100, 300, 50, 200);
     r.getcoord( &a, &b, &c, &d );                    // 用变量的指针做参数，带回多个结果
     cout << "left=" << a << '\t' << "top=" << b << endl;
     cout << "right=" << c << '\t' << "bottom=" << d << endl;
     r.print( );
     rr = r;                                          // 对象可整体赋值
     rr.print( );
}
```

此程序在类体外定义成员函数setcoord( )，在类体内定义成员函数getcoord( )和print( )。程序的运行结果是:

```
left=100        top=300
right=50        bottom=200
Area = 5000
Area = 5000
```

由前面的描述可知，类和对象具有“封装性”和“隐蔽性”。封装性是指将对象的静态属性和动态属性封装成一个整体——“类”类型；隐蔽性是指类的私有成员被隐藏在类内，在类外不可通过对象直接访问，从而达到保护数据成员的目的。

### 10.1.5 对象的存储空间

对象是一个类的变量，也称为类的实例。对象与结构体变量一样也占用存储空间，那么一个对象所占用的存储空间是多少呢？在VC++中，每定义一个对象，系统就给它分配空间，用于存放该对象具体的数据成员值；而成员函数的代码是相同的，所以系统将成员函数的存储空间处理成该类的所有对象共享同一代码空间。在例10.1中，一个Person类对象的数据成员所占用的存储空间理论上是 25个字节，但实际上 sizeof(a)和sizeof(Person)的值都是28，因为在VC++ 6.0编译系统中，内存空间分配为“取整”，一般以4个字节的整数倍分配空间。下面定义两个Person类对象，并给出它们占用的存储空间示意图如图10-1所示。

```
Person a, b, *pa=&a, *pb=&b;
a.SetData("Cheng", 'F', 20);
b.SetData("Zhang", 'M', 18);
```

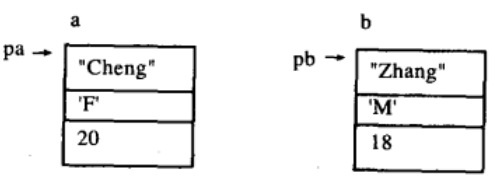



图10-1 对象的数据成员的存储空间

对象空间的分配和撤销是系统根据对象的作用域自动完成的，即进入对象作用域，系统自动为对象分配空间，退出对象作用域时，系统自动撤销对象空间。当编程者使用new和delete运算符动态建立和撤销对象时，对象空间的分配和撤销应由编程者控制完成。

如在程序执行的某一时刻动态申请了空间:

```
Person *p1, *p2;
p1 = new Person;      // 申请一个对象空间
p2 = new Person[10]; // 申请对象数组空间
```

那么，在程序执行的某一时刻，一般是程序执行结束前（或离开该指针的作用域前），就必须释放对象空间:

```
delete p1;
delete [ ]p2;
```

### 10.1.6 定义类和对象的有关说明

主要有以下几点需要说明:

（1）类中数据成员的类型可以是任意的，可以是基本数据类型（如整型、实型、字符型等）、构造数据类型（如数组类型、指针类型、引用类型、结构体类型等），当然也可以是新定义的“类”类型（结构体类型是“类”类型的特例）。自身类的对象不可以作为成员，而自身类的指针或引用可以作为类的成员。

```
class Obj
{
          int x;
          double y;
};
class Person
{
          char Name[20];
          char Sex;
          int  Age;
          Obj  obj1, obj2;          // 可以
          Person  a, b;             // 不可以
          Person  *pa;              // 可以
          ......
};
```

（2）当B类对象的指针或引用作为A类的成员时，如果B类的定义在A类之后，则在A类的定义之前，必须对B类做一个引用性说明。

```
class  B;                 // B类说明
class A                   // A类的定义
{
private:
          B  *pb;         // B类对象的指针pb是A类的数据成员
public:
          ......
};
class B                   // B类的定义
{
          ......
};
```

（3）与结构体类似，定义类的对象也有三种方法：

• 类的定义完成后，定义对象：

```
class A                        // A类的定义
{
    int x;
    float y;
public:
    ......
};
A  a1, a2, a3;                 // 定义A类对象
```

• 在定义类的同时定义对象：

```
class A                        // A类的定义
{
    int x;
    float y;
public:
```

```
    ......
} a1, a2, a3;                    // 定义A类对象
```

• 定义无类名的对象:

```
class                            // 无类名
{
    int x;
    float y;
public:
    ......
} a1, a2, a3;                    // 定义该类对象
```

这种方法的缺点是，只能定义一次该类对象，如果以后需要再次定义该类对象就不可以了。

(4) 在C++中结构体与类的区别是，结构体成员缺省的存取权限是公有的，而类中成员缺省的存取权限是私有的。结构体是类的特例。

## 10.2 初始化对象、撤销对象

结构体变量的成员赋初值可以在定义结构体变量的同时进行，如:

```
struct SPerson                   // 定义结构体类型
{
     char name[20];              // 姓名
     char sex;                   // 性别
     int  age;                   // 年龄
};
SPerson p1={ "Cheng", 'F', 20 };
```

而对于类，如果其成员是公有的，才能在定义对象时进行赋初值，如:

```
class  CRect
{
public:
   int left, top, right, bottom;
};
CRect r = {100, 300, 50, 200};
```

原因是: 结构体是类的特例，结构体成员的缺省存取权限是公有的。

而在定义类的对象时不可以通过下面的方法给类的私有数据成员赋初值，如:

```
class  CRect
{   int left, top, right, bottom;  };
CRect r = {100, 300, 50, 200};   // 非法的
```

在以前的例子中都是通过类的公有成员函数给对象的私有数据成员赋值的，使用起来比较麻烦。实际上C++提供了构造函数用于在定义对象时给对象的数据成员初始化。

**构造函数**和**析构函数**是类的两个特殊的成员函数。构造函数的功能是: 在创建对象时，由系统自动调用，用给定的值对数据成员初始化。析构函数的功能是: 系统在撤销一个对象时，自动调用它来做一些清理工作，它与构造函数的功能相反。请看下一节的详细描述。

### 10.2.1 构造函数和析构函数

#### 1. 构造函数

在定义一个类时，可以根据需要定义一个或多个构造函数（重载构造函数）。可以在类体内或类体外定义构造函数，在类体内定义构造函数的一般格式是:

```
ClassName(<形参列表>)  // ClassName是类名，在此处作为构造函数名
{ ..... }
```

在类体外定义构造函数的一般格式是:

```
ClassName :: ClassName(<形参列表>)
{ ..... }
```

构造函数的特点是：函数名与类名相同，而且函数没有返回值，构造函数只能由系统自动调用。在例10.5中，我们定义了四个重载的构造函数。

**例10.5** 定义日期类，利用构造函数初始化数据成员。程序放在头文件date.h中，如下所示:

```
#include <iostream.h>
class  Date
{
     int Year, Month, Day;
public:
Date( )                                    // 重载构造函数 1
{    Year=2003; Month=1; Day=1;  }
Date(int y)                                // 重载构造函数 2
{    Year=y; Month=1; Day=1;  }
Date(int y, int m)                         // 重载构造函数 3
{    Year=y; Month=m; Day=1; }
Date(int y, int m, int d)                  // 重载构造函数 4
{    Year=y; Month=m; Day=d; }
void ShowDate( )
{    cout <<Year<<'.'<<Month<<'.'<<Day<<endl;  }
};
```

主函数源文件为Li1005.cpp，内容如下:

```
#include  "date.h"
void main( )
{     Date d1;                             // 自动调用构造函数 1
      Date d2(2008);                       // 自动调用构造函数 2
      Date d3(2008, 10);                   // 自动调用构造函数 3
      Date d4(2008, 10, 6);                // 自动调用构造函数 4
      d1.ShowDate( );
      d2.ShowDate( );
      d3.ShowDate( );
      d4.ShowDate( );
}
```

在定义对象时，系统能自动根据所给参数的多少决定调用哪一个重载的构造函数。此程序运行结果是:

```
2003.1.1
2008.1.1
2008.10.1
2008.10.6
```

我们也可以定义带缺省值的构造函数，将上述构造函数简化，下述程序的功能与上述程序相同:

```
#include <iostream.h>
class  Date
{
     int Year, Month, Day;
public:
```

```
    Date(int y=2003, int m=1, int d=1)   //带参数缺省值的构造函数
    {     Year=y; Month=m; Day=d;        }
          void ShowDate( )
          {     cout <<Year<<'.'<<Month<<'.'<<Day<<endl;}
};
void main( )
{     Date d1, d2(2008), d3(2008, 10), d4(2008, 10, 6);
      d1.ShowDate( );
      d2.ShowDate( );
      d3.ShowDate( );
      d4.ShowDate( );
}
```

构造函数的特点如下:

(1) 构造函数是成员函数，函数体可写在类体内，也可写在类体外。

(2) 构造函数是特殊函数，该函数的名字与类名相同，不指定返回值类型。

(3) 构造函数可以重载，即可以定义多个参数个数不同的构造函数。

(4) 在创建对象时，系统自动调用构造函数。

### 2. 析构函数

在类体内定义析构函数的一般格式是:

```
~ClassName( ) // ClassName是类名，析构函数名为~ClassName
{ ..... }
```

在类体外定义析构函数的一般格式是:

```
ClassName :: ~ClassName( )
{ ..... }
```

析构函数的特点如下:

(1) 析构函数是成员函数，可在类体内定义，也可在类体外定义。

(2) 一般地，将析构函数定义成公有成员函数。

(3) 析构函数也是特殊函数，该函数的名字是类名前加 ~，用来与构造函数区分，该函数不指定返回值类型，没有参数。

(4) 一个类只能定义一个析构函数，即析构函数不允许重载。

(5) 析构函数可以被显式调用，也可以由系统自动调用。在下面两种情况下，析构函数会被自动调用:

- 当对象是系统自动创建的，则在对象的作用域结束时，系统自动调用析构函数。如在函数体内定义了一个局部对象，当函数执行结束时，系统要自动撤销该局部对象，在撤销该对象前，系统先自动调用析构函数，然后再撤销该对象的存储空间。若定义一个全局对象，则当程序结束时，该对象的析构函数被自动调用。
- 当一个对象是使用new运算符被动态创建的，在使用delete运算符释放它时，delete将会自动调用析构函数。

**例10.6** 定义学生类，利用构造函数初始化数据成员，利用析构函数做清理工作。

```
#include<iostream.h>
#include<string.h>
class Student
{
    char Num[10];                          //学号，注意: 用数组实现
    char *Name;                            //姓名，注意: 用指针实现
    int Score;                             //成绩
```

```
public:
    Student(char *nump, char *namep, int score)  //构造函数
    {    if(nump)                                //在构造函数中，不需要动态申请Num成员的空间
             strcpy(Num, nump);
         else
             strcpy(Num, "");
         if(namep)                               //在构造函数中，需动态申请Name成员的空间
         {    Name=new char[strlen(namep)+1];
              strcpy(Name, namep);
         }
         else Name=0;
         Score=score;
         cout<<"Constructor Called!\n";
    }
    ~Student( )                                  //在析构函数中，需释放Name成员的空间
    {    if(Name) delete [ ] Name;
         cout<<"Destructor Called!\n";
    }
    void Show( )
    {     cout << Num << '\t' << Name << '\t' << Score << endl;      }
};
void main( )
{  Student a("040120518", "George", 80);
   a.Show( );
}
```

此程序运行结果是:

```
Constructor Called!   //调用构造函数时的输出
040120518       George  80
Destructor Called!   //调用析构函数时的输出
```

### 3. 调用构造函数和析构函数的时机（对象的生存周期）

变量分为全局变量和局部变量，又分为动态变量和静态变量。对象实际上就是变量，所以对象也有全局对象和局部对象，动态对象和静态对象。我们知道，构造函数是在创建对象时系统自动调用的，而析构函数是在撤销对象时自动调用的，不同存储类别的对象调用构造函数和析构函数的时机不同:

（1）全局对象（总是静态的）——程序开始执行时，系统自动创建对象，调用构造函数；程序结束时，系统在撤销对象前调用析构函数。“程序开始执行时”是指main( )函数开始执行时。“程序结束时”是指main( )函数执行结束时。

（2）局部动态对象——当程序执行进入作用域时，在定义对象处系统自动创建对象，调用构造函数；退出作用域时，系统在撤销对象前调用析构函数。

（3）局部静态对象——当程序执行首次到达定义对象处时，系统自动创建对象，调用构造函数；程序结束时（即main( )函数执行结束时），系统在撤销对象前调用析构函数。

（4）动态申请的对象——使用 new 产生对象时，系统自动调用构造函数；使用 delete 撤销对象时，系统自动调用析构函数。

**例10.7**　调用构造函数和析构函数的时机。

```
#include <iostream.h>
class  Date
```

```
{
        int Year, Month, Day;
public:
        Date(int y=2002, int m=1, int d=1)          // 带参数的构造函数
        {     Year=y; Month=m; Day=d;
              cout<<"Constructor: ";
              ShowDate( );
        }
        void ShowDate( )
        {     cout <<Year<<'.'<<Month<<'.'<<Day<<endl;  }
        ~Date( )
        {     cout<<"Destructor: ";
              ShowDate( );
        }
};
Date d4(2008, 4, 4);                                  // 全局对象（静态的）
void fun( )
{
    cout << "进入 fun( )函数!\n";
    static Date d2(2008, 2, 2);                       // 局部静态对象
    Date d3(2008, 3, 3);                              // 局部动态对象
    cout << "退出 fun( )函数!\n";
}
void main( )
{
    cout << "进入 main( )函数!\n";
    Date d1(2008, 1, 1);                              // 局部动态对象
    fun( );
    fun( );
    cout << "退出 main( )函数!\n";
}
```

此程序运行结果是:

```
Constructor: 2008.4.4      // 调用构造函数，产生d4对象
进入 main( )函数!
Constructor: 2008.1.1      // 调用构造函数，产生d1对象
进入 fun( )函数!
Constructor: 2008.2.2      // 第一次进入fun( )函数，产生下述 d2、d3对象
Constructor: 2008.3.3
退出 fun( )函数!
Destructor: 2008.3.3       // 退出fun( )函数，撤销d3对象，不撤销d2对象
进入 fun( )函数!
Constructor: 2008.3.3      // 第二次进入fun( )函数，再次产生d3对象
退出 fun( )函数!
Destructor: 2008.3.3       // 退出fun( )函数，撤销d3对象
退出 main( )函数!
Destructor: 2008.1.1       // 退出main ( )函数，撤销d1、d2、d4对象
Destructor: 2008.2.2
Destructor: 2008.4.4
```

在主函数结束时，先撤销主函数中的局部对象，再撤销静态对象，对象的撤销顺序（即调用析构函

数的顺序）与定义顺序（即调用构造函数的顺序）相反。

### 10.2.2 缺省构造函数和缺省析构函数

#### 1. 缺省构造函数

在定义类时，如果没有定义构造函数，编译器就自动生成一个不带参数的缺省构造函数，其形式为:

```
ClassName :: ClassName( )  // ClassName是类名
{ }
```

**例10.8** 定义类时，不定义构造函数。

```
#include <iostream.h>
class  Date
{
    int Year, Month, Day;
public:
    void ShowDate( )
    {
          cout <<Year<<'.'<<Month<<'.'<<Day<<endl;
    }
};
void main( )
{
    Date d;                          //A
    d.ShowDate( );
}
```

则系统自动产生的缺省构造函数是:

```
Date::Date( )
{ }
```

在程序中定义对象时，如果不给初值，如程序中的A行，则系统自动调用缺省构造函数。本程序的运行结果是: -858993460.-858993460.-858993460，表示三个数据成员均没有被初始化，它们的初值是随机值。因为在缺省构造函数中没有做任何操作。

一般地，应该定义类的缺省构造函数。类的缺省构造函数有两种形式:

(1) 构造函数没有参数，如给上述例10.8增加下述缺省构造函数:

```
Date::Date( )                                  // 没有参数的构造函数，它是缺省构造函数
{     Year=2003; Month=1; Day=1;    }
```

此时程序的运行结果是: 2003.1.1。

(2) 构造函数的所有参数均带有缺省值，也称为缺省构造函数。如可为例10.8 定义构造函数:

```
Date(int y=2003, int m=1, int d=1)        // 每个参数均有缺省值，它也是缺省构造函数。
{     Year=y; Month=m; Day=d; }
```

此时程序运行的结果与第一种形式的缺省构造函数的效果一样，程序输出: 2003.1.1。

注意: (1) 在产生对象时，若不需要对数据成员进行初始化，可以不显式地定义缺省构造函数。(2) 在一个类的定义中，缺省构造函数只能有一个。也就是说，上述两种形式的缺省构造函数只能写一个。如果用户在编程时，定义了两个缺省构造函数，则程序编译时报错。(3) 若已经定义了构造函数（不论它是否为缺省构造函数），则编译系统不再自动生成缺省构造函数。

#### 2. 缺省析构函数

同理，在定义类时，如果用户没有定义析构函数，编译器就自动生成一个缺省析构函数，其格式如下:

```
ClassName :: ~ClassName( )
{ }
```

注意：（1）在撤销对象时，若不需要做任何结束工作，可以不显式地定义析构函数。（2）当类中有动态申请的数据空间时，必须显式地定义析构函数，撤销动态数据空间。如：例10.6中必须定义析构函数撤销 Name 指向的串空间。

### 10.2.3 拷贝构造函数和缺省拷贝构造函数

拷贝构造函数是一种特殊的构造函数，它的功能是用一个已知的对象来初始化一个被创建的同类对象。在类体外定义拷贝构造函数的格式如下：

```
ClassName :: ClassName( [const] ClassName & Obj )// 注意参数是本类对象的引用
{  <函数体>  }
```

Obj是实参对象的引用，const是类型修饰符，表示参数Obj在拷贝构造函数的函数体中是对象常量，不能被修改，以保护实参。当然亦可不对参数做const说明，即省略const，此时obj就不是常量了。

**例10.9** 定义一个“平面坐标点”类，测试拷贝构造函数的调用。

```
//头文件 "point.h"
class Point
{
    int x, y;
public:
    Point(int a=0, int b=0)                    //缺省构造函数
    {     x=a; y=b;  }
    Point(Point &p);                           //拷贝构造函数原型说明
    ~Point( )
    {    cout<<x<<','<<y<<" Destructor Called.\n" ;    }
    void Show( )
    {     cout<<"Point: "<<x<<','<<y<<endl;     }
    int Getx( )
    {     return x;}
    int Gety( )
    {     return y;}
};
Point::Point(Point &p)                         //定义拷贝构造函数
{  x=p.x; y=p.y;
   cout<<x<<','<<y<<" Copy-initialization Constructor Called.\n";
}
//文件 Li1009.cpp
#include <iostream.h>
#include "point.h"
void main( )
{  Point p1(6, 8), p2(4, 7);
   Point p3(p1);                               //A
   Point p4=p2;                                //B
   p1.Show( );
   p3.Show( );
   p2.Show( );
   p4.Show( );
}
```

在主函数的A行，调用拷贝构造函数，用已知对象p1初始化正在创建的新对象p3；在B行，调用拷贝

构造函数，用已知对象p2初始化正在创建的新对象p4。在程序结束时调用析构函数，按p4、p3、p2、p1的顺序撤销对象，这个顺序是建立对象的相反的顺序。

此程序的运行结果是：

```
6, 8 Copy-initialization Constructor Called.      //A行的输出
4, 7 Copy-initialization Constructor Called.      //B行的输出
Point: 6, 8
Point: 6, 8
Point: 4, 7
Point: 4, 7
4, 7 Destructor Called.                           //撤销 p4
6, 8 Destructor Called.                           //撤销 p3
4, 7 Destructor Called.                           //撤销 p2
6, 8 Destructor Called.                           //撤销 p1
```

拷贝构造函数的特点如下：

（1）该函数名与类名相同，因为它也是构造函数，并且该函数不被指定返回值类型。

（2）该函数只有一个参数，必须是本类对象的引用。

（3）每个类必须有一个拷贝构造函数，如果用户在定义类时没有定义拷贝构造函数，则系统自动产生一个缺省的拷贝构造函数。

如在例10.9中，如果用户不定义拷贝构造函数，则系统自动产生一个缺省的拷贝构造函数：

```
Point::Point(const Point &p)
{   x=p.x; y=p.y;  }
```

系统自动产生的拷贝构造函数，完成将参数对象的各个数据成员值逐个赋值给新产生的对象的各个数据成员。如果类的数据成员没有动态申请的数据空间时，可以不定义拷贝构造函数，利用系统自动产生的拷贝构造函数就可以了。但是，如果类的数据成员中有动态申请的空间，则必须定义拷贝构造函数，否则会出错。

**例10.10**　不定义拷贝构造函数时，运行出错。

```
//文件 Li1010.cpp
#include<iostream.h>
#include<string.h>
class Student
{
          char *Name;                          //姓名，注意：用指针实现
          int Age;                             //年龄
public:
          Student(char *namep, int age)        //构造函数
          {
               Age=age;
               if(namep)                       //在构造函数中，需动态申请空间
               {     Name=new char[strlen(namep)+1];
                     strcpy(Name, namep);
               }
               else Name=NULL;
          }
          ~Student( )                          //因在构造函数中动态申请了空间，故在析构函数
                                                 中，需释放空间
          {    if(Name) delete [ ] Name;   }
          void Show( )
```

```
        {    cout << Name << ',' << Age << endl;    }
};
void main( )
{
    Student a("George", 20);
    Student b=a;                                    //A
    b.Show( );
}
```

此程序运行时出错，原因是没有定义类的拷贝构造函数。系统自动产生的拷贝构造函数如下:

```
Student::Student(const Student &s)
{    Name = s.Name;                                 //注意: 直接赋地址值
     Age = s.Age;
}
```

注意: Name是指针，在程序的A行调用了拷贝构造函数后，两个对象及其存储空间的示意图如图10-2所示，假定字符串 "George" 的起始地址是1000:

对象a的指针成员Name和对象b的指针成员Name指向同一存储空间。程序结束时，先撤销对象b，调用析构函数将对象b的成员Name指针指向的串空间撤销；然后撤销对象a，由于a的成员Name指针指向的串空间已经被撤销了，再次撤销就出错了。正确的做法是，定义如下拷贝构造函数:

```
Student::Student(const Student &s)
{
     Age=s.Age;
     if(s.Name)
     {    Name = new char[strlen(s.Name)+1];  //C
          strcpy(Name, s.Name);
     }
     else Name=NULL;
}
```

此时，在主函数的A行调用拷贝构造函数后，两个对象及其存储空间的示意图如图10-3所示。

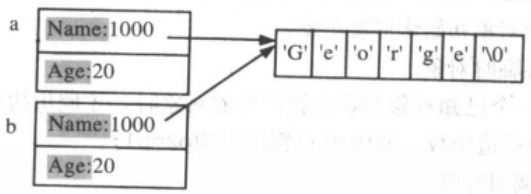



图10-2 对象“浅”拷贝结果

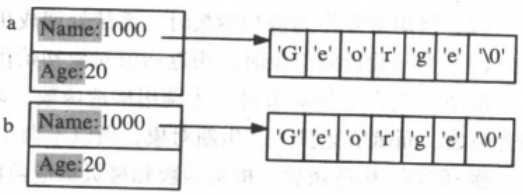



图10-3 对象“深”拷贝结果

注意: 在拷贝构造函数中，将字符串"George"复制了一份，假定新的串空间的起始地址是2000（就是在上述C行新分配的串空间的起始地址），则b的Name指针值是2000，在程序运行结束，撤销对象b和a时，分别释放串空间，就不会出错了。

何时必须定义拷贝构造函数呢？分两种情况:

(1) 在产生对象时，若只需拷贝同类型对象的部分数据成员。

(2) 若对象中的某个数据成员是使用 new 运算符动态地申请存储空间时，则必须在类中显式地定义一个完成拷贝功能的构造函数，以便实现数据成员的复制。同时应显式定义相应的析构函数，撤销动态分配的空间，如例10.10中的情况。

### 10.2.4 拷贝构造函数的调用时机

例10.11 在本例中，使用例10.9中“平面坐标点”类的头文件point.h，测试用对象做函数参数及函

数返回值时拷贝构造函数的使用。

```
#include <iostream.h>
#include "point.h"
Point move(Point p, int xoffset, int yoffset)  // 普通函数，不是类的成员函数
{  int x = p.Getx( )+xoffset, y = p.Gety( )+yoffset;
   Point t(x, y);
   return t;
}
void main( )
{  Point p1(6, 8), p2;
   p2=move(p1, 2, 4);
   p2=p1;// 赋值时，不调用构造函数，只有产生新对象时，才调用构造函数
}
```

此程序的运行结果是:

```
6,8 Copy-initialization Constructor Called.      // A
8,12 Copy-initialization Constructor Called.     // B
8,12 Destructor Called.                          // 撤销对象t
6,8 Destructor Called.                           // 撤销对象p
8,12 Destructor Called.                          // 撤销内存临时对象
6,8 Destructor Called.                           // 撤销对象p2
6,8 Destructor Called.                           // 撤销对象p1
```

上述A行输出，是在主函数中调用函数move( )时产生的，用实参p1初始化形参p，相当于Point p=p1，此时调用拷贝构造函数。上述B行输出，是在函数返回时，用对象t初始化内存临时对象时的输出。假定该内存临时对象的名字是temp，相当于 Point temp=t。为什么有一个临时对象呢？因为函数返回对象时，系统处理成：首先在内存中建立一临时对象，暂时存放返回对象的值；接着，撤销函数move( )中的局部对象t、p；然后将临时对象temp的值赋给对象p2；最后撤销临时对象temp。

通过例10.9、例10.10和例10.11可以看出，在三种情况下，系统自动调用拷贝构造函数:

(1) 明确表示由一个对象初始化另一个对象时，例如：例10.9 中的A、B两行。

(2) 当用对象作为函数参数时，系统处理成用实参对象初始化形参对象。

(3) 当函数返回对象时，用返回值对象初始化内存临时对象。

注意：当产生新对象时，才调用构造函数。即用一个已知对象初始化新产生的对象时，才调用拷贝构造函数。而赋值运算不产生新对象，所以不调用拷贝构造函数。如例10.11程序中的p2=p1;。

**例10.12** 构造函数、析构函数和拷贝构造函数的调用时机。

```
#include <iostream.h>
class  Sample
{
    int x, y;
public:
   Sample( int a=0, int b=0)
   {    x=a; y=b;
        cout<<x<<','<<y<<" default constructor\n";
   }
   Sample( Sample &s )
   {    x=s.x; y=s.y;
        cout<<x<<','<<y<<" copy constructor\n";
   }
```

```
    ~Sample( )
    {    cout<<x<<','<<y<<" destructor\n";  }
    void add( )
    {    x+=10; y+=10;  }
};
void main( )
{
    Sample a (1, 2), b (3, 4), c (5, 6);
    Sample sarr[3]={a, b, c};                    //A
    Sample *ptr=sarr;                            //B
    for(int i=0; i<3; i++, ptr++)
          ptr->add( );                           //C
    cout<<"main end\n";
}
```

程序运行结果如下:

```
1, 2 default constructor
3, 4 default constructor
5, 6 default constructor
1, 2 copy constructor
3, 4 copy constructor
5, 6 copy constructor
main end
15, 16 destructor
13, 14 destructor
11, 12 destructor
5, 6 destructor
3, 4 destructor
1, 2 destructor
```

在A行定义了一个对象数组，用已有的三个对象来初始化对象数组的三个元素，调用了三次拷贝构造函数。在B行定义了一个Sample类对象的指针，并让其指向对象数组。在C行，通过指向对象数组的指针调用数组元素的成员函数。注意本程序各构造函数和析构函数的调用顺序。

### 10.2.5 利用构造函数进行类型转换

C++中不同的数据类型可以相互赋值。一个类定义完成，就产生了一个新的数据类型。新数据类型的变量（对象）与已有数据类型的变量可以相互赋值。若已定义了复数类Complex，则若有Complex c; double x, y;，则可能出现下述两种操作：(1) c=x; 将实部为x、虚部为0的一个复数赋值给复数c。(2) 希望将一个x为实部、y为虚部的复数赋值给c。根据C++中的类型转换原则，即将等号右边的运算结果转换成与等号左边类型一致的数据，再进行赋值：C++应该把上述两种情况处理成 c=Complex(x) 和 c=Complex(x, y)，于是只要定义相应的构造函数，就能实现上述两种情况。如果出现 x=c的情况，可参见本书13.2.3节的详细描述。

**例10.13** 利用构造函数完成类型转换。

```
//文件 Li1013.cpp
#include <iostream.h>
class Complex
{
    double Real, Image;
public:
    Complex(double x=0, double y=0)
```

```
    {   Real=x; Image=y;
        Show( );
        cout << "调用了构造函数\n";
    }
    ~Complex( )
    {   Show( );
        cout << "调用了析构函数\n";
    }
    void Show( )
    {     cout<<'('<<Real<<','<<Image<<')';     }
};
void main( )
{  Complex c1(3, 5), c2;                    //A
   c1 = 8.0;                                //B
   c2 = Complex(9.0, 9.0);                  //C
}
```

此程序的运行结果是:

```
(3, 5)调用了构造函数          // 在A行创建c1对象时，调用构造函数
(0, 0)调用了构造函数          // 在A行创建c2对象时，调用构造函数
(8, 0)调用了构造函数
(8, 0)调用了析构函数          // 创建、撤销临时对象，调用构造函数、析构函数
(9, 9)调用了构造函数
(9, 9)调用了析构函数          // 创建、撤销临时对象，调用构造函数、析构函数
(9, 9)调用了析构函数          // 在程序结束，撤销c2对象时，调用析构函数
(8, 0)调用了析构函数          // 在程序结束，撤销c1对象时，调用析构函数
```

编译器将B行处理成: c1= Complex(8.0);，在等号右边，调用构造函数产生临时复数对象，将该临时对象赋值给c1后撤销临时对象。编译器对C行的处理与B行是一样的。

## 10.3 成员函数的特性

### 10.3.1 内联函数和外联函数

类的成员函数可以分为内联函数和外联函数。在类体内定义的成员函数是内联函数，在类体外定义的函数是外联函数。在类体外也可以定义内联函数，方法是在函数定义的首部增加inline修饰。如: 简化的复数类定义如例10.14所示。

**例10.14** 在类体外定义内联函数。

```
class Complex
{
    double Real, Image;
public:
    Complex(double x=0, double y=0);              //A 在类体内，给出成员函数说明
    void Show( );                                 //B 在类体内，给出成员函数说明
};
inline Complex::Complex(double x, double y)       //C 在类体外定义内联函数
{   Real=x; Image=y;   }
inline void Complex::Show( )                      //D 在类体外定义内联函数
{   cout<<'('<<Real<<','<<Image<<')';  }
```

注意: 程序中A行在说明构造函数时给出了参数的缺省值，因此在C行定义构造函数时就不必给出参数的缺省值了。

C++处理一般函数的调用过程是：先保护程序的执行现场，然后流程转去执行被调函数的函数体，执行完再转回到函数调用处，恢复现场，执行其后的语句。而C++处理内联函数的方法是：编译时在调用函数处用内联函数的代码来替换函数调用语句，这样程序执行时不需要函数调用（函数调用和返回时，需要保存和恢复程序执行的现场），减少了程序执行的时间开销，提高了程序的执行速度。但是程序的总代码空间加大。因此一般将函数体较短，需频繁循环调用的函数定义成内联函数。这样，函数代码空间不会膨胀得很厉害，同时程序的执行效率也会提高。

将内联函数与前面讲过的宏定义进行比较，它们的代码效率是一样的，但内联函数要优于宏定义，因为内联函数遵循函数的类型和作用域规则，它与一般函数更相近。

内联函数一定要在调用之前定义，并且内联函数无法递归调用。

### 10.3.2 成员函数的重载

前面讲过构造函数可以重载，一般的成员函数也可以重载，我们通过一个例子来说明。

**例10.15** 处理一个由数组构成的线性表，动态产生线性表，并输出线性表中的数据。

说明：线性表用来存放若干个整数，用一个指针ListPtr指向该线性表的第0个元素，nMax表示线性表的最大空间（即最多可以存放的整型元素个数），nElem表示当前线性表中实际存放的元素个数。线性表是动态产生的，最初通过构造函数初始化线性表，对线性表可以进行的主要操作有：向线性表中加入元素、修改线性表某元素值、输出线性表全体元素值等，在向线性表加入元素时如果已有的存储空间不够了，则自动扩大线性表空间。程序如下：

```
#include <iostream.h>
class  ListClass
{
    int  *ListPtr;                          // 指向线性表的指针
    int  nMax;                              // 线性表的最大长度
    int  nElem;                             // 线性表中当前元素的个数
public:
    ListClass(int n=10)                     // 构造函数，初始化线性表，最大长度的缺省值为10
    {     nElem=0;
          nMax=n;
          if(n)
              ListPtr = new int[n];
          else
              ListPtr=0;
    }
    ~ListClass(void)                        // 析构函数
    {    delete [nMax] ListPtr;}
    int  Elem(int);                         // 重载函数①，在线性表尾增加一个元素
    int &Elem(unsigned n)                   // 重载函数②，返回线性表中第n个元素的引用
    {    return ListPtr[n];}
    int Elem(void)                          // 重载函数③，返回线性表中当前元素的个数
    {    return nElem;  }
    int Max(void)                           // 返回线性表的长度
    {  return nMax; }
    int GetElem(int i)                      // 返回线性表第i个元素的值
    {     if((i>=0) && (i<nElem)) return ListPtr[i];
          else { cout << "下标越界" << endl; return(0); }
    }
    void Print(void);                       // 输出线性表中的所有元素，在类体外实现
};
```

```
int ListClass::Elem(int elem)                    // 重载函数①，在线性表尾增加一个元素
{
        if(nElem<nMax)                           // 如果实际元素个数小于表的长度
        {   ListPtr[nElem++]=elem;               // 增加元素
            return(nElem);                       // 返回新表的元素个数
        }
        else                                     // 如果实际元素个数等于表的长度
        {       int *newptr;
                newptr=new int[nMax+10];         // A 申请一新表空间
                for(int i=0; i<nElem; i++)
                     newptr[i]=ListPtr[i];       // 将原表中的元素拷贝到新表中
                delete[nMax]ListPtr;             // 释放原表空间
                nMax+=10;
                ListPtr=newptr;
                ListPtr[nElem++]=elem;           // 增加元素
                return(nElem);                   // 返回新表的元素个数
        }
}
void ListClass::Print(void)                      // 输出线性表中的所有元素
{       for(int i=0; i<nElem; i++)
                cout << ListPtr[i] << '\t';
        cout << endl;
}
void main(void)
{   ListClass  list(6);
    for(int i=0; i<5; i++)
          list.Elem(i);                          // 调用重载函数①
    cout<<"线性表的长度为: "<<list.Max( )<<endl;
    cout<<"线性表的元素个数为: ";
    cout<<list.Elem( )<<endl;                    // 调用重载函数③
    cout<<"线性表的元素为: ";
    list.Print( );
    list.Elem(3u)=100;                           // 调用重载函数②
    cout<<"线性表的第3个元素的值为: ";
    cout<<list.Elem(3u)<<endl;                   // 调用重载函数②
    list.Elem(20);                               // 调用重载函数①
    list.Elem(200);                              // 调用重载函数①
    cout<<"现在线性表的长度为: "<<list.Max( )<<endl;
    cout << "现在线性表中的元素个数为: ";
    cout <<list.Elem( )<<endl;                   // 调用重载函数③
    cout<<"线性表的元素为: ";
    list.Print( );
    cout << "线性表的最后一个元素为: ";
    cout<<list.GetElem(list.Elem( )-1)<<endl;    // 调用重载函数③
}
```

程序中的A行，一次性增加10个元素空间。目的是减少反复给线性表分配新空间的概率。反复给线性表分配新空间的情况发生在线性表已满，但此时还要连续给线性表增加元素之际。

此程序的运行结果是:

```
线性表的长度为: 6
```

```
线性表的元素个数为: 5
线性表的元素为: 0    1    2    3    4
线性表的第3个元素的值为: 100
现在线性表的长度为: 16
现在线性表中的元素个数为: 7
线性表的元素为: 0    1    2    100    4    20    200
线性表的最后一个元素为: 200
```

上述程序中的重载函数①，可以简化成用下述方法实现:

```
int ListClass::Elem(int elem)                    // 重载函数①，在线性表尾增加一个元素
{   if(nElem==nMax)                              // 线性表空间不够了
    {   int *newptr;
        newptr=new int[nMax+10];                 // 申请一新表空间
        for(int i=0; i<nElem; i++)
            newptr[i]=ListPtr[i];                // 将原表中的元素拷贝到新表中
        delete[nMax]ListPtr;                     // 释放原表空间
        nMax+=10;
        ListPtr=newptr;                          // 让指针指向新表空间
    }
    ListPtr[nElem++]=elem;                       // 增加元素
    return(nElem);                               // 返回新表的元素个数
}
```

## 10.4 构造函数和对象成员

在定义一个类时，可以把一个已定义的类的对象作为该类的成员。产生新定义类的对象时，须对它的对象成员进行初始化，且只能通过其对象成员的构造函数来实现。

**例10.16** 初始化对象成员。

```
#include <iostream.h>
#include <math.h>
class Point                              // 定义“点” 类
{
    int x, y;
public:
    Point(int a=0, int b=0)
    {    x=a; y=b;
         cout<<x<<','<<y<<" 构造 Point\n";
    }
    int Getx( )
    {    return x;}
    int Gety( )
    {    return y;}
    ~Point( )
    {    cout<<x<<','<<y<<" 析构 Point\n";    }
};
class Line                               // 定义“线” 类，两点决定一条线
{
    int width, color;                    // 指定“线” 的宽度、颜色
    Point p1, p2;                        // 指定“线” 的两个端点
public:
```

```
    Line(int x1, int y1, int x2, int y2, int w, int c) : p1(x1,y1), p2(x2,y2)   //A
    {    width=w; color=c;
         cout<<width<<','<<color<<"构造 Line\n";
    }
    double LineLen( )
    {    double len;
         int x1, y1, x2, y2;
         x1=p1.Getx( ); y1=p1.Gety( );  x2=p2.Getx( ); y2=p2.Gety( );
         len=sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
         return(len);
    }
    ~Line( )
    {    cout<<width<<','<<color<<"析构 Line\n"; }
};
void main( )
{  Line Li(0, 0, 1, 1, 3, 6);
   cout<<"长度="<<Li.LineLen( )<<endl;
}
```

此程序的运行结果是:

```
0, 0 构造 Point
1, 1 构造 Point
3, 6构造 Line
长度=1.41421
3, 6析构 Line
1, 1 析构 Point
0, 0 析构 Point
```

说明: 注意上述A行，在Line类的构造函数后加一个冒号，在冒号后面调用对象成员p1和p2的构造函数。从运行结果可以看出，先调用对象成员的构造函数，再调用类自身的构造函数；析构函数的调用顺序正好相反。

在类的定义中，说明对象成员的一般格式为:

```
class  ClassName
{
      ClassName_1   c1;
      ClassName_2   c2;
        ......
      ClassName_n   cn;
public:
      ClassName(args) : c1 (arg_1), c2 (arg_2),..., cn (arg_n)   //A
      {......}
......
};
```

其中，ClassName_1，ClassName_2，...，ClassName_n是已经定义的类名。为了初始化对象成员c1，c2，...，cn，类ClassName的构造函数形如上述格式中的A 行。我们把参数表后冒号后用逗号隔开的对c1，c2，...，cn初始化的列表称为成员初始化列表，其中的参数表arg_1，arg_2，...， arg_n依次为调用相应成员所在类的构造函数时应提供的实参列表。这些参数通常来自于ClassName构造函数的形参args。

注意: (1) args中的形参必须带有类型说明，而arg_1，arg_2，...，arg_n是实参，不需要类型说明，它们可以是来自于args的变量，也可以是常数或表达式等。(2) 调用对象成员构造函数的顺序与写在成员

初始化列表中的顺序无关，而与对象成员的定义顺序有关，先定义的先调用。如将例10.16中的A行改写为:

```
Line(int x1, int y1, int x2, int y2, int w, int c) : p2(x2, y2), p1(x1, y1)  // A
```

仍然是先调用p1(x1,y1)，再调用p2(x2,y2)，因为p1的定义在前，p2的定义在后。

## 10.5 this指针

this是一个隐含于每一个类的成员函数中的特殊指针。该指针是一个指向正在被某一个函数操作的对象的指针。当对象调用成员函数时，自动将对象自身的地址(指针)，传递给成员函数，在成员函数中可直接使用该指针，指针名为this。如有一个类:

```
#include <iostream.h>
class  Sample
{
    int x, y;
public:
    Sample( int a=0, int b=0)
    { x=a; y=b; }
    void Print( )
    {     cout << x <<'\n';
          cout << y <<'\n';
    }
};
void main( )
{    Sample c2(3, 5);
     c2.Print( );
}
```

在该例中，成员函数中隐含着一个指针this，它指向调用成员函数的对象，在成员函数中可以直接使用该指针，如下述程序与上述程序等价:

```
#include <iostream.h>
class  Sample
{
    int x, y;
public:
   Sample( int a=0, int b=0)
   {  this->x = a;                        // 在此例中this = &c2
      this->y = b;
   }
   void Print( )
   {   cout << (this->x) <<'\n';          // 在此例中this = &c2
       cout << (this->y) <<'\n';
   }
};
void main( )
{   Sample c2(3, 5);
    c2.Print( );
}
```

例10.17 说明this指针的使用。

```
#include <iostream.h>
class  Sample
```

```
{
    int x, y;
public:
    void fun(int a, int b)
    {
          this -> x=a;
          this -> y=b;
          cout<<this<<'\t'<<x<<'\t'<<y<<endl;    //E
    }
};
void main( )
{
    Sample c1, c2;
    cout << &c1 <<'\n';                          //A
    c1.fun(1, 2);                                //B
    cout << &c2 <<'\n';                          //C
    c2.fun(3, 4);                                //D
}
```

此程序的运行结果是:

```
0x0065FDF0                          // 主函数中A行输出的是c1的地址
0x0065FDF0    1       2             // 在B行通过c1调用成员函数fun( )时，E行的输出
0x0065FDE8                          // 主函数中C行输出的是c2的地址
0x0065FDE8    3       4             // 在D行通过c2调用成员函数fun( )时，E行的输出
```

说明: 在A行输出的地址值与在B行通过c1调用成员函数fun( )时，E行输出的地址值相等，说明 this = = &c1。在C行输出的地址值与在D行通过c2调用成员函数fun( )时， E行输出的地址值相等，说明 this = = &c2。在定义类成员函数时，并不知道 this 指向哪一个对象，只有当某一对象调用该成员函数时，才知道 this 指向的具体对象。所以，两个对象调用同一个成员函数，在E行输出的地址值this是不同的。

## 习题10

### 一、选择题、填空题

1. 类成员缺省的访问特性是_____，结构体成员缺省的访问权限是_____。

   A）protected　　B）public　　C）private　　D）friend

2. 下列关于类的成员的访问特性的描述中，正确的是_____。

   A）必须首先说明具有私有特性的成员

   B）数据成员必须说明成私有的，成员函数必须说明成公有的

   C）每个成员前都必须有标明访问特性的关键字

   D）在同一个类中，说明访问特性的关键字可以多次使用

3. 关于类和对象，下列说法中不正确的是_____。

   A）类与对象的关系类似于数据类型与变量的关系

   B）对象是类的一个实例

   C）任何一个对象必定属于一个特定的类

   D）一个类只能有一个对象

4. 关于类的成员函数，下列说法正确的是 _____。

   A）必须在类体内定义　　B）一定是内联函数

   C）不可以重载　　D）可以设置参数的缺省值

5. 以下_____不可以作为类的成员。

A) 自身类对象的指针　　B) 自身类的对象

C) 自身类对象的引用　　D) 另一个类的对象

6. 已知一个类，类名为A。可以定义A类的对象或定义A类的指针，下列一定不正确的形式为_____。

A) `A a1 ;`　B) `A a2(16) ;` C) `A a3( ) ;`　D) `A *p = new A ;`

7. 已知一个类Complex，有下述两行:

```
Complex c ;      // A
Complex c ();    // B
```

以下描述中，错误的是_____。

A) A行定义了一个对象，并调用缺省构造函数。

B) B行定义了一个对象，并调用缺省构造函数。

C) B行是一个函数原型说明，函数名为c，函数没有参数，返回值类型是Complex。

D) A行和B行两行的意义不一样。

8. 设有下列对象定义:

```
class A
{
public:
     int a, b;
} a1={1, 2}, a2, a3;
class
{
public:
     int a, b;
} a4;
```

则以下正确的定义或赋值形式是_____。

A) `a3=a2=a1;` B) `a4=a1` C) `A *p=&a4;` D) `A &re=a4;`

9. 下列描述中，_____不是构造函数的特征。

A) 构造函数可以重载

B) 必须显式为类定义一个构造函数

C) 无须为构造函数指定返回值类型

D) 构造函数在产生对象时由系统自动调用

10. 下列有关析构函数的描述中，正确的是_____。

A) 析构函数的定义只能在类体内

B) 析构函数可以有一个或多个参数

C) 析构函数是在撤销对象时，系统自动调用的

D) 析构函数可以重载

11. 以下关于构造函数和析构函数的描述中，错误的是_____。

A) 若用户不定义缺省构造函数，则系统自动生成一个缺省构造函数

B) 若用户不定义拷贝构造函数，则系统自动生成一个拷贝构造函数

C) 若用户不定义析构函数，则系统自动生成一个析构函数

D) 以上A错误，而B、C正确

12. 定义复数类如下:

```
class Complex
{    double real, image;
```

```
public:
      complex(double r=0, double i=0){ real=r; image=i; }
      complex(complex &c){       real=c.real; image=c.image; }
};    若有：  Complex c1;          //A
      Complex c2(3, 5);          //B
      Complex c3(c2);            //C
      c2=c1;                     //D
```

则下述说法中正确的是_____。

A）C行和D行均调用了拷贝构造函数　　B）C行调用了拷贝构造函数

C）B行调用了拷贝构造函数　　D）A行调用了拷贝构造函数

13.
```
class A
{    int x, y;
public:
      A(int a, int b)     { x=a; y=b; }
      void show( ){ cout<<x<<','<<y<<endl; }
      void set(int a, int b){ x=a; y=b; }
};
void main( )
{    A obj;
     obj.set(3, 5);
     obj.show( );
}
```

对上述程序段，以下说法中，正确的是_____。

A）编译时报错，程序无法运行　B）编译无错，运行正常，输出 3, 5

C）编译无错，运行时报错　　D）编译时报警告错，但运行正常，输出 3, 5

14. 在下列程序中，C类的数据成员中有一个A类对象和一个B类对象。

```
#include <iostream.h>
class A
{   int a;
public:
    A(int x=10){ a=x; }
    int GetA( ){ return a; }
};
class B
{   int b;
public:
    B(int x){ b=x; }
    int GetB( ){ return b; }
};
class C
{   int c;
    A aobj;
    B bobj;
public:
    C(int x, int y, int z) : aobj(y),bobj(z)   // E
    { c=x; }
    void Print( )
    { cout<<aobj.GetA( )<<'\t'<<bobj.GetB( )<<'\t'<<c<<endl; }
};
```

```
void main( )
{
    C cobj(3, 6, 8);
    cobj.Print( );
}
```

程序在E行定义了构造函数，如果将该行改为下列中_____选项时，便会出现编译错误。

A）C(int x, int y, int z): aobj(z), bobj(y)

B）C(int x, int y, int z): bobj(z)

C）C(int x, int y, int z): aobj(y)

D）C(int x, int y, int z): aobj(x), bobj(y+z)

## 二、找错、运行结果

1. 指出下列程序的错误:

```
#include <iostream.h>
class A
{    int x, y;
public:
     A( ){ x=y=0; }
     A(int i=0, int j=0) { x=i; y=j; }
     void show( ){ cout<<x<<','<<y<<endl; }
};
void main( )
{
     A a1, a2(6, 8);
     a1.show( );
     a2.show( );
}
```

2. 写出下列程序的运行结果:

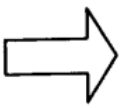
```
#include <iostream.h>
class A
{   int x, y;
public:
     A( ){ x=y=0; cout<<"Default constructor called.\n"; }
     A(int i, int j)     { x=i; y=j; cout<<"Constructor called.\n";}
     ~A( ){ show( ); cout<<"Destructor called.\n"; }
     void show( ){ cout<<x<<','<<y<<endl; }
};
void main( )
{    A a1, a2(6, 8);
     a1.show( );
     a2.show( );
}
```

3. 根据以下两个方框中的内容，写出两个版本的运行结果:

```
#include <iostream.h>
class A
{     int x, y;
public:
     A(int i=0, int j=0) { x=i; y=j; Show( ); cout<<"Constructor called.\n"; }
     A(A &a);
```

```
    ~A( ){ Show( ); cout<<"Destructor called.\n"; }
    void Set(int i=0, int j=0) { x=i; y=j;}
    void Show( ){ cout<<x<<','<<y<<','; }
};
A::A(A &a)
{ x=a.x; y=a.y; Show( ); cout<<"Copy constructor called.\n"; }
```

```
A fun(A a)
{
    a.Show( );
    cout<<endl;
    a.Set(3, 7);
    return a;
}
```



```
A & fun(A & a)
{
    a.Show( );
    cout<<endl;
    a.Set(3, 7);
    return a;
}
```

```
void main( )
{
    A a1, a2(6, 8), a3(a2), a4;
    a4=fun(a3);
    a4.Show( );
    a3.Show( );
    cout<<endl;
}
```

4. 写出下列程序的运行结果:

```
#include <iostream.h>
class A
{    int x, y;
public:
    A(int i=0, int j=0) { x=i; y=j; Show( ); cout<<"Constructor called.\n"; }
    ~A( ){ Show( ); cout<<"Destructor called.\n"; }
    void Set(int i=0, int j=0) { x=i; y=j;}
    void Show( ){ cout<<x<<','<<y<<','; }
};
A &fun(A &a)
{    a.Show( );
    cout<<endl;
    a.Set(3, 7);
    return a;
}
void main( )
{    A a1, a2(6, 8), a3(a2), a4=a1;
    a4=fun(a3);
    a4.Show( );
    cout<<endl;
}
```

5. 写出下列程序的运行结果:

```
#include <iostream.h>
class Complex
{    int Real, Image;
public:
```

```
    Complex(int x=0, int y=0)
    {   Real=x; Image=y;
        Show( );
        cout<<"Construct\n";
    }
    ~Complex( )
    {   Show( );cout<<"Destruct\n";      }
    void Show( )
    {     cout<<'('<<Real<<','<<Image<<')';   }
    void Print( )
    {     Show( ); cout<<endl; }
};
void main( )
{   Complex c[5]={3, Complex(4), Complex( ), Complex(5, 6)};
    for(int i=0; i<5; i++)      c[i].Print( );
        cout<<"Exit main( )\n";
}
```

6. 写出下列程序的运行结果:

```
#include <iostream.h>
class Test
{
    int x;
public:
    Test(int i=0)
    { x=i; }
    void Print( )
    {     cout<<x<<','<<this<<endl;    }
};
void main( )
{   Test t1(8), t2(9);
    cout<<&t1<<','<<&t2<<endl;        //A
    t1.Print( );                      //B
    t2.Print( );                      //C
}
```

若A行的输出是0x0012FF7C、0x0012FF78，则B行和C行的输出是_____。

7. 写出下列程序的运行结果:

```
#include <iostream.h>
class List
{
    int a[10], n;
public:
    List(int a[10], int n)
    {
        List::n = n;          //List::n表示数据成员n，以便与参数n区别
        for(int i=0; i<n; i++)
            List::a[i] = a[i];
    }
    void Copy(List L)
    {   if(this == &L)
```

```
            cout<<"Same! not copy!\n";
        else
        {   n=L.n;
            for(int i=0; i<n; i++)
                a[i] = L.a[i];
            cout<<"Different! copied!\n";
        }
    }
    void Show( )
    {   for(int i=0; i<n-1; i++) cout<<a[i]<<',';
        cout<<a[i]<<endl;
    }
};
void main( )
{   int a1[10]={1, 2, 3}, a2[10]={4, 5, 6, 7, 8};
    List L1(a1, 3), L2(a2, 5);
    L1.Show( );
    L2.Show( );
    L2.Copy(L1);
    L1.Show( );
    L2.Show( );
    L1.Copy(L1);
    L2.Copy(L2);
    L1.Show( );
    L2.Show( );
}
```

## 三、编程题

在编程题中，若不做特别说明，成员函数均为公有成员函数。

1. 定义一个复数类Complex，数据成员包括实部和虚部。成员函数包括：① 设置实部。② 设置虚部。③取实部值。④ 取虚部值。⑤ 输出复数。在主函数中定义一个复数类对象，然后对全体成员函数进行测试。
2. 定义一个三维坐标体系下的线段类Line，数据成员包括两个端点的坐标。成员函数包括：① 设置第一个端点坐标。② 设置第二个端点坐标。③ 求出线段的长度。④ 取出第一个端点坐标，参数为三个指针，分别指向一个端点的x、y、z坐标。⑤ 取出第二个端点坐标，参数为三个引用，分别为另一个端点的x、y、z坐标变量的引用。⑥ 输出两个端点的坐标以及线段的长度。在主函数中定义一个线段类对象，然后对全体成员函数进行测试。
3. 定义一个平面坐标体系下的三角形类Triangle，数据成员包括三个顶点的坐标。要求：将三角形的三个顶点定义成对象数组。成员函数包括：① 设置顶点1坐标。② 设置顶点2坐标。③ 设置顶点3坐标。④ 求任意两点之间的距离，函数原型为double distance(int i, int j)，i和j表示顶点号，它们的取值范围是1~3，例如调用distance(1, 2)表示求顶点1和顶点2之间的距离。⑤ 求三角形周长。⑥ 求三角形面积。⑦ 输出三角形的三个顶点坐标、三条边长度、周长和面积。⑧ 将三角形转换成关于Y轴镜像对称的三角形（Y轴坐标不变，X轴坐标取反）。⑨ 将三角形转换成关于X轴镜像对称的三角形（X轴坐标不变，Y轴坐标取反）。⑩ 将三角形在平面上平移，函数原型为void move(int xoff, int yoff)，参数xoff 和yoff分别表示三角形在X轴和Y轴上的平移的相对距离，可以是正数或负数。在主函数中定义一个三角形类对象，设置其各成员值，输出有关该三角形的全部信息。然后将该三角形分别做X轴镜像转换、Y轴镜像转换以及平移转换，每转换一次，输出转换后的三角形的全部信息。
4. 定义一个日期类Date，数据成员有年（year）、月（month）、日（day）。成员函数有：① 私有成员函数int isleap( )判断year是否是闰年，若是则返回1，若不是则返回0。② 私有成员函数int check( )判断日期

是否合法，若合法则返回1，若不合法则返回0。③ 设置年、月、日值，并检测是否为合法日期。④ 按yyyy/mm/dd格式输出日期，若是闰年，还要输出是否是闰年的信息。在主函数中定义一个日期类对象，任意输入年、月、日值，然后对全体成员函数进行测试。

5. 定义一个描述学生的类Student，数据成员包括学号（num）、姓名（name）、性别（sex）、家庭住址（faddr）、联系电话（tel）、email地址（email）以及数学（math）、物理（physics）、外语（eng）、程序设计（prog）四门课的成绩。对于数据成员，要求用字符数组实现学号、姓名、家庭住址、联系电话、email地址，用字符型量实现性别（'f'表示女性，'m'表示男性），用整型量实现四门课的成绩。成员函数包括：① 设置学号。② 设置姓名。③ 设置性别。④ 设置家庭住址。⑤ 设置联系电话。⑥ 设置email地址。⑦ 设置数学、物理、外语、程序设计四门课的成绩，函数原型是void setscore(char tag, int score)，当tag的值分别是'm'、'p'、'e'和'c'时，分别设置数学、物理、外语、程序设计四门课的成绩。⑧ 输出一个学生的全部描述信息。在主函数中定义一个学生类对象，然后对全体成员函数进行测试。
6. 定义一个描述教师的类Teacher，数据成员包括工号（num）、姓名（name）、性别（sex）、家庭住址（faddr）、联系电话（tel）、email地址（email）、职务（headship）、职称（post）和工资（salary）。对于数据成员，要求用字符数组实现工号、姓名、家庭住址、联系电话、email地址、职务和职称，用字符型量实现性别，用整型量实现工资。成员函数包括：① 设置工号。② 设置姓名。③ 设置性别。④ 设置家庭住址。⑤ 设置联系电话。⑥ 设置email地址。⑦ 设置职务。⑧ 设置职称。⑨ 设置工资。⑩ 输出一个教师的全部描述信息。在主函数中定义一个教师类对象，然后对全体成员函数进行测试。
7. 为第1题复数类增加构造函数：① 无参的构造函数，它是缺省构造函数。② 带参数的构造函数，参数为实部和虚部的初值，参数无缺省值。③ 拷贝构造函数。在主函数中增加测试三个构造函数的语句。
8. 为第2题线段类增加构造函数：① 缺省构造函数，它带参数，参数为两个端点的坐标，参数具有缺省值。② 拷贝构造函数。在主函数中增加测试这两个构造函数的语句。
9. 在本题中，将一个类的对象作为另一个类的数据成员。

   定义一个平面坐标体系下的“点”类Point，数据成员为x，y坐标值（整型量）。成员函数包括：① 带参数的缺省构造函数。② 拷贝构造函数。③ 置x，y坐标值。④ 取x，y坐标值，参数为两个整型量的引用，分别用于获取x，y坐标值。

   定义一个平面坐标体系下的“三角形”类Triangle，数据成员为三个Point类的对象p1、p2和p3。成员函数包括：① 带参数的缺省构造函数，参数为整型量x1, y1, x2, y2, x3, y3，分别是三角形三个顶点的坐标，各参数均有缺省值。② 带参数的构造函数，参数是三个Point类对象的引用。③ 设置顶点1坐标，参数为两个整型变量。④ 设置顶点2坐标，参数为两个整型变量。⑤ 设置顶点3坐标，参数是一个Point类对象的引用。⑥ 取顶点1坐标，参数为两个整型变量的指针，即将两个坐标值放入两个指针指向的变量中。⑦ 取顶点2坐标，参数为两个整型变量的引用，即将两个坐标值赋值给两个引用变量中。⑧ 取顶点3坐标，参数是一个Point类对象的引用。⑨ 求三角形周长。⑩ 求三角形面积。⑪ 输出三角形的三个顶点坐标、周长和面积。

   在主函数中定义一个Triangle类对象和若干Point类对象，测试Triangle类的所有成员函数。
10. 用一个动态数组模拟堆栈操作。初始时堆栈为空，将下标为0的元素看成栈底，一般栈底是固定的。对堆栈的常规操作有：① 入栈：即向堆栈中压入数据（即将数据加入到数组中），加入数据时栈顶升高。② 出栈：从栈顶取出一个数据。一个数据出栈后，栈顶降低。如图10-4所示。

    图中size表示栈的大小，*n*表示当前栈中已有的元素个数。在做入栈操作时，要考虑若堆栈已满，要动态申请另一个新的栈空间（一次比原来多10个元素），将原栈内容拷入新栈，然后释放原栈空间。在做出栈操

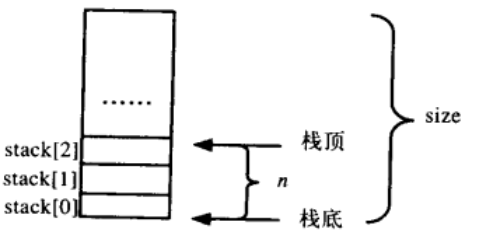



图10-4 堆栈示意图

作时，要考虑若栈中无元素，则给出警告信息。

定义一个类Stack，实现堆栈及相关操作。其中数据成员有：① `int *stack;` 栈底指针。② `int n;` 栈中已有元素的个数。③ `int size;` 表示栈的大小。成员函数有：① 构造函数，参数为栈的初始大小，缺省值为10。需动态申请栈空间。② `void push(int x);` 将x入栈，可能要动态申请栈空间。③ `int pop( );` 栈顶元素出栈。④ `int stacksize( );` 返回栈的大小。⑤ `int stacknum( );` 返回栈中实际元素的个数。⑥ 析构函数，释放栈空间。在主函数中定义一个Stack类对象，任意输入若干整数，进行入栈出栈等操作，测试类的所有成员函数。

# 第11章 类和对象的其他特性

## 11.1 静态成员

通常，一个类的不同对象，其数据成员的存储空间是相互独立的。也就是说，一个对象的成员与另一个对象的成员占用不同的存储空间。如果将类的一个成员定义成静态型的，则该类的所有对象的该成员共用同一存储空间。静态成员的提出是为了解决对象之间的数据共享问题。静态数据成员是类的所有对象共享的成员，而不是某个对象的成员。在类中，静态成员分为静态数据成员和静态成员函数。

### 11.1.1 静态数据成员

下面介绍静态数据成员的定义、初始化及引用。

**(1) 定义**：静态数据成员的定义是在类型说明符前加关键字static。例如:

```
class Sample
{
private:
     int a, b, c ;          // 定义非静态数据成员
     static int d ;         // 定义静态数据成员
      ......
}
```

**(2) 初始化**：静态数据成员的初始化与一般数据成员的初始化不同，必须在类体外进行，格式如下:

```
<类型标识符> <类名>::<静态数据成员名>=<值>;
```

如对上述静态数据成员d的初始化应在类体外进行，形式为: `int Sample::d = 10;`。

注意：必须在类体外对静态数据成员进行初始化，初始化时，前面不需要加static，但应在静态数据成员名前加类名限定，以免与一般的全局静态变量混淆。

**(3) 引用**：示例如下。

**例11.1** 静态数据成员的使用。

```
#include <iostream.h>
class Sample
{
    int n;
    static int sum;
public:
    Sample(int x) { n=x; }
    void add( ){ sum+=n; }
    void disp( )
    {     cout<<"n="<<n<<",sum="<<sum<<endl;   }
};
int Sample::sum = 0;                              // 静态数据成员初始化
void main( )
{       Sample a(2), b(3), c(5);
        a.add( );    a.disp( );
        b.add( );    b.disp( );
        c.add( );    c.disp( );
```

```
        // A
        cout <<"sizeof(a) : "<<sizeof(a)<<endl;
        cout <<"sizeof(b) : "<<sizeof(b)<<endl;
        cout <<"sizeof(c) : "<<sizeof(c)<<endl;
}
```

当程序执行到A处时，请看a、b和c三个对象及类的静态成员的存储状况。只有成员n是属于对象的，而成员sum是属于类的，三个对象共享sum空间，这一点可以从程序的运行结果中得到印证（见图11-1）。

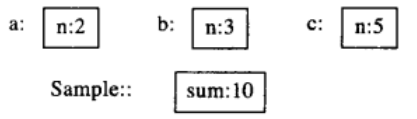



图11-1　类的静态成员

此程序的运行结果是：

```
n=2, sum=2
n=3, sum=5
n=5, sum=10
sizeof(a) : 4
sizeof(b) : 4
sizeof(c) : 4
```

如果将主函数改写为：

```
void main( )
{  Sample a(2), b(3), c(5);
     a.add( );  b.add( );  c.add( );
     a.disp( ); b.disp( );  c.disp( );
}
```

此程序的运行结果是：

```
n=2, sum=10
n=3, sum=10
n=5, sum=10
```

### 11.1.2　静态成员函数

静态成员函数与静态数据成员一样，它们都是属于类的静态成员，而不是属于对象的成员。因此，对静态成员函数的引用必须用类名限定，格式如下：

```
<类名>::<静态成员函数名>
```

在静态成员函数的实现中，可以引用类中的静态成员，但不能直接引用类中的非静态成员。如果想在静态成员函数中引用非静态成员，可通过参数对象来引用。

**例11.2**　静态成员函数的使用。

```
#include <iostream.h>
class Sample
{
    int a;
    static int b;
public:
    Sample(int x) { a=x; b+=x;}
    static void disp(Sample s)                    //A
    {      cout<<"a="<<s.a<<", b="<<b<<endl;  } //B
};
int Sample::b=10;                                 //静态数据成员赋初值
void main( )
{   Sample s1(2), s2(3);
```

```
    Sample::disp(s1);                    //C
    Sample::disp(s2);                    //D
}
```

在A行定义了一个静态成员函数，并用一对象做参数。在B行，注意a和b的引用方式不同，a是非静态成员，不可直接访问；而b是静态成员，可直接访问。注意在C行和D行调用静态成员函数的方法。程序的运行结果如下：

```
a=2, b=15
a=3, b=15
```

## 11.2 友元

### 11.2.1 友元函数

前面讲过，类具有封装和信息隐蔽的特性。也就是说，只有在类的成员函数中才能直接访问私有成员。在类外不可直接访问私有成员。在类外只能通过公有函数接口才能访问私有成员。而调用公有函数接口是需要时间开销的，且在编写程序时也比较烦琐，如下例所示。

**例11.3** 通过公有函数接口访问私有成员。

```
#include <iostream.h>
#include <math.h>
class Point
{
    int x, y;
public:
    Point(int a=0, int b=0)
    {     x=a; y=b;      }
    ~Point( ){ }
    void Show( )
    {     cout<<"Point("<<x<<','<<y<<")\n";    }
    int Getx( )
    {     return x;      }
    int Gety( )
    {     return y;      }
};
double distance(Point &p1, Point &p2 )    // 注意: 此函数是普通函数，求两点之间的距离
{
    return sqrt((p1.Getx( )-p2.Getx( ))*(p1.Getx( )-p2.Getx( ))
                +(p1.Gety( )-p2.Gety( ))*(p1.Gety( )-p2.Gety( )));
}
void main( )
{   Point p1, p2(1, 1);
    cout<<distance(p1, p2)<<endl;
}
```

对于上例，在distance( )函数中多次调用公有函数接口访问私有数据成员，这样带来的函数调用时间开销是比较大的。在普通函数中能否直接访问类的私有的或保护的成员？回答是肯定的，但是，必须将该普通函数定义为类的友元函数。方法是：

```
class  A
{   ...
    friend void fun(...);     // 说明函数fun( )是类A的友元函数
```

```
    ...
};
void fun(...)
{  class A a;
    ...
}
```

在类A的定义中将函数fun( )定义为类A的友元函数。于是在fun( )函数中，类A的对象a可直接访问它的私有的或保护的数据成员。可将例11.3改写成如下形式：

```
#include <iostream.h>
#include <math.h>
class Point
{
    int x, y;
public:
            Point(int a=0, int b=0)
            {     x=a; y=b;       }
            ~Point( ){ }
            void Show( )
            {     cout<<"Point("<<x<<','<<y<<"\n";      }
            int Getx( )
            {     return x;       }
            int Gety( )
            {     return y;       }
            friend double distance(Point&, Point& );                    //A
};
double distance(Point &p1, Point &p2 )
{
return sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));      //B
}
void main( )
{  Point p1, p2(1,1);
   cout<<distance(p1,p2)<<endl;
}
```

程序在A行说明distance( )函数是Point类的友元函数，因此，在B行就可以直接访问Point类对象的私有数据成员。使用友元函数，使程序变得较简洁，而且省去了多次调用成员函数带来的时间开销。

另外，我们还可以将main函数作为一个类的友元函数。

**例11.4**　将main函数作为一个类的友元函数。

```
#include <iostream.h>
class Sample
{
    int x, y;
public:
            Sample(int a=0, int b=0)
            {    x=a; y=b;                    }
            void Show( )
            {    cout<<"x="<<x<<", y="<<y<<"\n";       }
            friend void main( );              // 说明主函数是类的友元函数
};
```

```
void main( )
{           Sample s(3, 6);                    // main( )函数是Sample类的友元函数,
            cout<<"x="<<s.x<<", y="<<s.y<<"\n"; // 所以可直接访问类Sample的私有成员。
            s.Show( );
}
```

程序的运行结果如下:

```
x=3, y=6
x=3, y=6
```

关于友元函数的几点说明:

(1) 友元函数不是类的成员函数。因此，对友元函数指定访问权限无效，可以把友元函数的说明放在private、public、protected 的任意段中。

(2) 使用友元函数的目的是提高程序的运行效率。

(3) 慎用友元函数，因为它可在类外直接访问类的私有或保护成员，破坏了类的信息隐藏的特性。

### 11.2.2 一个类的成员函数可作为另一个类的友元函数

**例11.5** 在本例中，将一个类的成员函数说明为另一个类的友元函数。

```
#include <iostream.h>
class N;                                    // N类的引用性说明。
                                            // 因为N的定义在M之后，而M中使用了N，如下面程序中的A行。
class M
{
    int a, b;
public:
    M(int x, int y){ a=x; b=y;}
    void Print( )
    {    cout<<"a="<<a<<'\t'<<"b="<<b<<endl;  }
    void Setab(N &);                        // Setab( )是类M的成员函数 // A
} ;
class N
{
    int c, d;
public:
    N(int a, int b){ c=a; d=b;}
    void Print( )
    {     cout<<"c="<<c<<'\t'<<"d="<<d<<endl;          }
    friend void M::Setab(N&);               // 将M类的成员函数说明成本类的友元函数
};
void  M::Setab(N &obj)                      // M的成员函数Setab( )是类N的友元函数,
{
       a=obj.c;                             // 因此在Setab( )中可直接访问类N的私有成员。
       b=obj.d;
}
void main(void)
{  M    m(25, 40);
              N  n(55, 66);
       cout<<"m: "; m.Print( );
       cout<<"n: "; n.Print( );
       m.Setab(n);
       cout<<"m: "; m.Print( );
}
```

程序的运行结果如下:

```
m: a=25 b=40
n: c=55 d=66
m: a=55 b=66
```

在类M中只有在成员函数Setab( )中可以直接访问类N的私有成员，类M的其他成员函数均不可以直接访问类N的私有成员。

### 11.2.3 友元类

友元还可以是类，即一个类可以作为另一个类的友元。当一个类作为另一个类的友元时，就意味着这个类的所有成员函数都是另一个类的友元函数。可按如下方式定义友元类:

```
class A
{     ...
     friend class  B;          // B类是A类的友元
} ;
class B
{   ...
public:
     void f1( );          ┐
     float f2( );         ├   // 因此，在B类的所有成员函数中，
     float f3( );         ┘   // 均可直接访问A类的私有成员。
};
```

**例11.6** 将一个类说明为另一个类的友元类。

```
#include <iostream.h>
class B;                                        // 引用性说明
class A
{
          int x;
          void Disp( )                          // 私有成员函数
          {     cout<<"x="<<x<<endl;}
          friend B;                             // 定义B是A的友元类
} ;
class B
{
public:
          void Set(int n)
          {     A a;
                a.x = n;                        // 可访问A类对象的私有数据成员x
                a.Disp( );                      // 可访问A类对象的私有成员函数disp( )
          }
};
void main(void)
{      B b;
       b.Set(3);
}
```

程序输出 x=3 。

## 习题11

### 一、选择题

1. 下面各函数，_____不是类的成员函数。

A）构造函数　　B）析构函数

C）友元函数　　D）拷贝初始化函数

2. 下述有关静态数据成员的描述中，_____是错误的。

A）说明静态数据成员时，前面要加修饰符static

B）静态数据成员要在类体外进行初始化

C）引用静态数据成员时，要在静态数据成员名前加<类名>和作用域运算符

D）静态数据成员不是所属类的所有对象共有的

3. 以下有关友元函数的描述中，错误的是_____。

A）友元函数不是成员函数

B）友元函数加强了类的封装性

C）在友元函数中可以访问所属类的私有成员

D）友元函数的作用是提高程序的运行效率

4. 关于成员函数特征的下述描述中，_____是错误的。

A）成员函数一定是内联函数　　B）成员函数可以重载

C）成员函数可以设置参数的缺省值　　D）成员函数可以是静态的

**二、写出以下程序的运行结果**

```
#include <iostream.h>
class Count
{
    static int count;
public:
    Count( ){ count++; cout<<count<<'\t'; }
    static int GetC( ) { return count; }
    ~Count( ){ count- -; cout<<count<<'\t'; }
};
int Count::count;
void main( )
{
    Count c1, c2, c3, c4;
    cout<<endl<<Count::GetC( )<<endl;
    cout<<"Exit main!\n";
}
```

**三、编程题**

为第10章的“习题10”中的编程题第1题复数类增加一个友元函数，用于输出一个复数对象。友元函数的原型为：`void Output( Complex & );`。

# 第12章 继承和派生

## 12.1 继承的基本概念

在本书面向对象部分的前两章，我们重点介绍了面向对象程序设计的第一个特性——封装性。在本章，我们介绍第二个特性——继承性。继承性提供了在已有类的基础上扩展出新类的机制，可以节省重复代码的编写工作，是软件重用的基础。

例如已定义了一个类A，它具有一些属性（用数据成员描述）和行为（用成员函数实现），现在可以对A类加以扩展，即增加一些属性和行为，构成一个新类B，此时B类将A类已有的属性和行为继承下来。也就是说在定义B类时，对继承了的A类的已有的属性和行为的代码不需要重新书写，只需对新增加的属性和行为的代码进行书写，这样就可以减少很多代码编写工作量。我们称A类为“基类”或“父类”，B类为“派生类”或“子类”。A派生了B，或称B继承了A。派生类是对基类的扩充，参见图12-1。

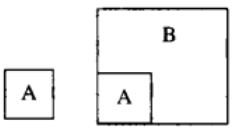


图12-1 派生类是对基类的扩充

在C++中，有两种继承：单一继承和多重继承。当派生类仅由一个基类派生时，称为单一继承；当派生类由多个基类派生时，称为多重继承。如图12-2a为单一继承，有三个类“人员类”、“学生类”和“教师类”。人员类具有一些公共的属性，如姓名、年龄、性别、身高、体重等；学生类除了上述公共属性外，还有学号、所学专业、课程、课程的成绩等属性；同理，教师类除了上述公共属性外，还有工作部门、职工号、专业、讲授课程等属性。在图12-2a中存在着两个单一继承关系，第一个是“学生类”继承“人员类”，第二个是“教师类”继承“人员类”。“学生类”继承“人员类”，就是在定义学生类时，只要增加定义学生类的属性，而基类的属性被继承下来，此时学生类具有姓名、年龄、性别、身高、体重、学号、所学专业、课程、课程的成绩等属性，是一个“完整”的类。又如图12-2b为多重继承，“日期类”具有年、月、日等属性，“时间类”具有时、分、秒等属性，“日期时间类”多重继承日期类和时间类，就是将两个基类的属性“组合”出一个完整的日期时间类。在日期时间类中，虽然没有增加属性，却具有年、月、日、时、分、秒属性。

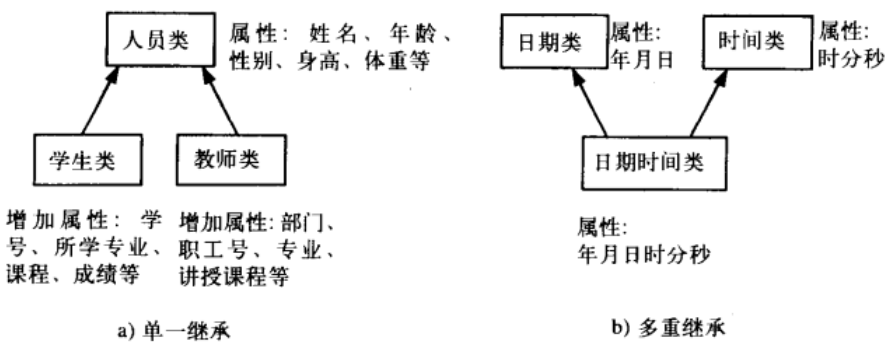


图12-2 单一继承和多重继承

## 12.2 单一继承

在单一继承方式下，定义派生类的一般格式为：

```
class <派生类名>: <继承方式> <基类名>
{
      [private:     // 私有成员说明
       ...]
      [public:      // 公有成员说明      } 定义派生类新成员
       ...]
      [protected:   // 保护成员说明
       ...]
} ;
```

继承方式有三种，它们是public——公有继承(派生)，private——私有继承(派生)，protected——保护继承(派生)。这三种继承方式有什么区别呢？下面分别介绍。

### 12.2.1 公有继承(派生)

公有继承(派生)的定义如下，它的继承关系如图12-3所示。

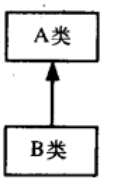



图12-3 单一公有继承

```
class A
{
      ......
};
class B: public A // 公有继承A
{
      ......
};
```

在公有继承时，B类继承了A类的所有成员，A类中的public和protected成员在B类中仍然是public和protected成员，而A类中的private成员虽然被B类继承下来了，但在B类中不可直接访问。请看表12-1。

**表12-1 不同继承方式下基类和派生类成员的访问权限**

| 继承方式 | 在基类A中 | 在派生类B中 | 在B类外(如通过类B的对象) |
|---|---|---|---|
| public （公有） | public成员 | 仍为public成员 | 可直接访问 |
| | protected成员 | 仍为protected成员 | 不可直接访问 |
| | private 成员 | 不可直接访问 | 不可直接访问 |
| private（私有） | public成员 | 变为private成员 | 不可直接访问 |
| | protected成员 | 变为private成员 | 不可直接访问 |
| | private 成员 | 不可直接访问 | 不可直接访问 |
| protected（保护） | public成员 | 变为protected成员 | 不可直接访问 |
| | protected成员 | 仍为protected成员 | 不可直接访问 |
| | private 成员 | 不可直接访问 | 不可直接访问 |

从表12-1中可以看出，不论何种继承方式，基类的private成员，在派生类中均不可直接访问。只有公有继承的基类的公有成员，在派生类中可直接访问。在公有继承的派生类中，保持基类的public成员和protected成员的属性。在私有继承的派生类中，将基类的public成员和protected成员的属性均变成了私有成员。在保护继承的派生类中，将基类的public成员变成了保护成员，而protected成员仍然保持其属性。

例如，有定义如下：

```
class A
{
private:
   int ax;
protected:
   int ay;
```

```
public:
    int az;
    int  f1( ){...};
    void f2( ){...};
    int  Getax( ){return ax;}
};
class B: public A
{
private:
    int bx;
protected:
    int by;
public:
    int bz;
    void f3( ){...};
    float f4( ){...};
};
```

B类继承A类后，将A类的成员继承为自己的成员，相当于对B类做如下定义：

```
class B
{
        private:  int  ax;        //M
private:
    int bx;
protected:
    int ay;
    int by;
public:
    int az;
    int  f1( ){...};
    void f2( ){...};
    int  Getax( ){return ax;}
    int bz;
    void  f3( ){...};
    float f4( ){...};
};
```

注意B类的斜体黑体部分是从A类继承下来的成员，可把它们当成B类“自己的”成员使用。但是注意M行，虽然A类的ax成员被继承下来了，但在B类中却不可以直接访问它，如果需要访问它，还必须通过A类的公有成员函数接口。如果A类已定义好，用继承的方式定义B类，在定义B类时，可以省掉A类已有的成员函数的代码书写，但是却好像对B类也写了这些代码，这就是继承的优点，代码的重用是系统帮您完成的。

**例12.1**　先定义“点”类Point，再由“点”类派生出“圆”类Circle。

```
#include <iostream.h>
#define PI 3.14159
class Point                                          //定义“点”类
{
    int x, y;                                        //“点”的坐标
public:
    Point(int a=0, int b=0)
    {     x=a; y=b;  }
```

```
    void ShowPoint( )
    {     cout<<"Point:("<<x<<','<<y<<")\n";     }
    int Getx( )
    {     return x;}
    int Gety( )
    {     return y;}
    void Setxy(int a, int b)
    {     x=a; y=b;  }
};
class Circle: public Point                          //A  定义“圆”类
{
    int r ;                                         // “圆”的半径
public:
    Circle(int x, int y, int ra) : Point(x, y)  //B
    {    r = ra;       }
    void Setr(int ra)
    {    r = ra;       }
    double Area( )                                  //求圆的面积
    {    return PI*r*r;  }
    void Move(int x_offset, int y_offset)           //将圆心坐标平移
    {    int x1=Getx( );                            //C
         int y1=Gety( );                            //D
         x1+=x_offset; y1+=y_offset;
         Setxy(x1, y1);                             //E
    }
    void ShowCircle( )
    {     ShowPoint( );                             // F
          cout<<" Radius: "<<r<<'\t';
          cout<<"Area: "<<Area( )<<endl;            //G
    }
};
void main( )
{   Circle c(1, 1, 1);
    c.ShowCircle( );
    c.Move(1, 2);
    c.ShowCircle( );
    c.Setxy(4, 5);                                  // 重新置圆心坐标  //H
    c.Setr(2);                                      // 重新置半径值
    c.ShowCircle( );
}
```

程序中A行表示Circle类公有继承Point类。B行冒号后的Point(x, y)的作用是在派生类的构造函数中调用基类的构造函数，以初始化基类的x、y两个私有成员。基类中的公有成员函数Getx( )、Gety( )、Setxy( )、ShowPoint( )被继承下来，变成Circle类自身的公有成员函数。所以在C、D、E、F行可以直接调用它们，就像在G行中调用Circle类本身的新定义的成员函数Area( )一样。在主函数中定义了一个Circle类对象c，在使用它时，感觉Circle类是一个完整的类，就像Point类不存在一样。面向对象程序设计中继承性的优点就在于此，系统提供了相当多的已定义好的基类，用户可以根据具体的应用，在已有类的基础上构架新类，在派生类中，可以直接使用父类部分的代码，但却不需要重新编写父类的代码。这样可以加快软件开发的速度，保障软件开发的质量。继承性是软件重用的基础。

仅仅就本例而言，如果采用独立建立Circle类的方法，即Circle类的定义不是建立在Point类的基础之

上，那么在定义Circle类时，必然要包括类似于Point类的代码。独立建立Circle类的方法与用继承的方法建立Circle类比较，好像没有省去多少代码，但是试想如果没有Point类，若要独立建立圆类、长方形类、正方形类、三角形类等等，那么每一个类都具有类似于Point类的操作代码，在各个独立的类中均需要重复书写与Point类相似的代码。有了继承性后，可以将类似的操作代码放在基类Point中，各个派生类只要继承公共操作即可，省掉了多次重复代码的编写工作。

### 12.2.2 私有继承(派生)

私有继承(派生)的定义如下，它的继承关系如图12-4所示。

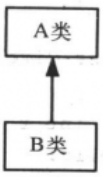



图12-4　单一私有继承

```
class A
{
    ......
};
class B: private A     // 私有继承A
{
    ......
};
```

从表12-1中我们可以看出，在私有继承时，基类中的public和protected成员在派生类中变成了private成员，而基类中的private成员虽然被派生类继承下来了，但在派生类中不可直接访问。

例如，有定义如下:

```
class A
{
private:
   int ax;
protected:
   int ay;
public:
   int az;
   int  f1( ){...};
   void f2( ){...};
   int  Getax( ){return ax;}
};
class B: private A
{
private:
   int bx;
protected:
   int by;
public:
   int bz;
   void f3( ){...};
   float f4( ){...};
};
```

则相当于对B类做如下定义:

```
class B
{
   private: int ax;      //M
private:
   int bx;
```

```
    int ay;
    int az;
    int  f1( ){...};
    void f2( ){...};
    int  Getax( ){return ax;}
protected:
    int by;
public:
    int bz;
    void  f3( ){...};
    float f4( ){...};
};
```

注意B类的斜体黑体部分是从A类继承下来的成员，可把它们当成B类“自己的”私有成员使用。同样注意M行，虽然A类的ax成员被继承下来了，但在B类中却不可以直接访问它，如果需要访问它，还必须通过A类的公有成员函数接口。

如果将例12.1的A行改写成 class Circle : private Point，就变成了私有继承，那么基类Point中的所有公有成员函数继承到了Circle类中，均变成了私有成员函数。此时，程序中H行通过派生类对象访问私有成员函数Setxy( )就是非法的。

由于私有继承用得不多，故此在这里不再详细叙述。

### 12.2.3 保护继承(派生)

从表12-1中我们还可以看出，在保护继承时，基类中的public和protected成员在派生类中为protected成员，而基类中的private成员虽然被派生类继承下来了，但在派生类中不可直接访问。

### 12.2.4 private成员和protected成员的区别

我们知道对一个单独的类来说，private成员和protected成员都是在类内可以被直接访问，在类外不可以被直接访问，没有什么区别。但是学习了类的继承特性以后，我们可以了解一下它们的区别。

无论何种继承方式，private成员均无法在派生类中被直接访问。而对于protected成员，根据不同的派生方式，protected成员的直接访问特性可以被（或不被）传递到派生类的派生类中。

（1）对于公有派生，基类的protected成员在派生类中依然保持protected属性。因此，protected成员的直接访问特性可以被传递到派生类的派生类中。

（2）对于私有派生，基类的protected成员在派生类中变成了private成员，此时基类的protected成员就再也无法在派生类的派生类中被直接访问了。在继承或派生链中，一旦出现私有继承，则父类的成员的“类内直接访问特性”就无法在后面的派生中传递下去。

（3）对于保护派生，基类的protected成员在派生类中依然保持protected属性，此时情况与公有派生类似。

类中protected成员的优点是：既可以在本类中实现数据的隐藏（在类外不可被直接访问），又可以将其类内直接访问特性传递到派生类中（在派生类中可直接访问）。但private成员只能实现本类中的数据隐藏，而不能将其类内直接访问特性传递到派生类中。

## 12.3 多重继承

由多个基类派生出一个类，称为多重继承。一般格式为：

```
class <派生类名>: <继承方式1> <基类名1>, <继承方式2> <基类名2>,...
{                                          //以下定义派生类新成员
      [private:  // 私有成员说明
       ... ]
```

```
    [public:    // 公有成员说明
    ... ]
    [protected: // 保护成员说明
     ... ]
} ;
```

**例12.2** 先定义“点”类Point和“半径”类Radius，再由Point类和Radius类多重派生出“圆”类Circle。

```
#include <iostream.h>
#define PI 3.14159
class Point
{
protected:                                                          //A
     int x, y;
public:
     Point(int a=0, int b=0){ x=a; y=b; }
     void ShowPoint( ){ cout<<"Point:("<<x<<','<<y<<")\n";}
     int Getx( ) { return x; }
     int Gety( ) { return y; }
     void Setxy(int a, int b){ x=a; y=b; }
};
class Radius
{
protected:                                                          //B
     int r;
public:
     Radius(int ra=0){ r = ra; }
     void Setr(int ra){ r = ra; }
     int Getr( ){ return r; }
     };
class Circle : public Point, public Radius                          //C
{
public:
     Circle(int x, int y, int ra) : Point(x, y), Radius(ra)         //D
     { }
     double Area( )
     { return PI*r*r; }                                             //E
     void Move(int x_offset, int y_offset)
     { x+=x_offset; y+=y_offset; }                                  //F
     void ShowCircle( )
     {     ShowPoint( );
           cout<<"Radius: "<<r<<'\t';
           cout<<"Area: "<<Area( )<<endl;
     }
};
void main( )
{    Circle  c(1, 1, 1);
     c.ShowCircle( );
     c.Move(1, 2);
     c.ShowCircle( );
     c.Setxy(4, 5);
     c.Setr(2);
```

```
    c.ShowCircle( );
}
```

本例中的继承关系如图12-5所示。

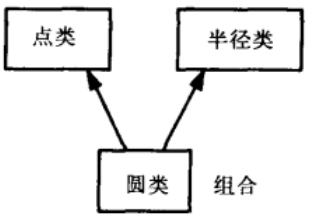


图12-5 多重公有继承

注意在本程序中A行和B行，将基类的数据成员定义成保护成员，这样既可以在基类中隐藏数据成员，又能在公有派生类的成员函数中直接访问这些数据成员，如程序中的E行和F行，这样处理比在例12.1中派生类通过基类的公有成员函数访问基类私有成员方法简单、快捷。在C行，可以看出Circle是由两个基类Point和Radius公有派生出来的。在D行，Circle类的构造函数本身没有代码，它通过调用基类的构造函数来初始化基类的数据成员。

程序的运行结果为:

```
Point:(1, 1)
Radius: 1       Area: 3.14159
Point:(2, 3)
Radius: 1       Area: 3.14159
Point:(4, 5)
Radius: 2       Area: 12.5664
```

## 12.4 基类成员的初始化

### 12.4.1 基类的构造函数和析构函数的调用顺序

实际上，在例12.2的D行，我们已经看到，若一个类是由多个基类派生出来的，则在定义派生类构造函数时，应调用基类的构造函数，初始化基类成员。

派生类构造函数的一般格式是:

```
ClassName :: ClassName(args) : Base1(arg1), Base2(arg2), ..., Basen(argn)
{  <派生类自身的构造函数体>  }
```

其中ClassName是派生类名，Base1，Base2，...，Base*n*为基类的类名。args是派生类自身的构造函数的形参列表，arg1，arg2，...，arg*n*是调用基类构造函数的实参列表。

派生类构造函数的执行顺序为：先依次调用基类构造函数Base1( )，Base2( )，...，Base*n*( )，最后执行<派生类自身的构造函数体>。

析构函数的调用顺序为：先执行派生类自身的析构函数体，然后按 ~Base*n*( )，...，~Base2( )，~Base1( )的顺序调用基类的析构函数。

**例12.3** 多重继承中基类构造函数和析构函数的调用顺序。

```
#include <iostream.h>
class Base1
{
protected:
    int data1;
public:
    Base1(int a=0)
    {     data1 = a;
          cout<<"Base Constructor1\n";
    }
    ~Base1( )
    {     cout<<"Base Destructor1\n";  }
};
```

```
class Base2
{
protected:
    int data2;
public:
    Base2(int a=0)
    {     data2 = a;
          cout<<"Base Constructor2\n";
    }
    ~Base2( )
    {     cout<<"Base Destructor2\n";  }
};
class Derived : public Base1, public Base2                          //A
{
    int d;
public:
    Derived(int x, int y, int z) : Base1(x), Base2(y)               //B
    {  d=z; cout<<"Derived Constructor\n";}
    ~Derived( )
    {  cout<<"Derived Destructor\n";  }
    void Show( )
    {  cout<<data1<<','<<data2<<','<<d<<endl;  }
};
void main( )
{  Derived  c(1, 2, 3);
   c.Show( );
}
```

程序的运行结果是:

```
Base Constructor1
Base Constructor2
Derived Constructor
1, 2, 3
Derived Destructor
Base Destructor2
Base Destructor1
```

实际上，调用基类的构造函数的顺序并不是B行的书写顺序决定的，而是A行定义派生类时的派生顺序决定的，即使将B行改写成Derived(int x, int y, int z) : Base2(y), Base1(x)，程序的运行结果依然不变。

### 12.4.2　对象成员构造函数和析构函数的调用顺序

若派生类中包含对象成员，则在派生类的构造函数的初始化成员列表中不仅要列举基类的构造函数，而且要列举对象成员的构造函数。

**例12.4**　对象成员构造函数和析构函数的调用顺序。

```
#include <iostream.h>
class Base1
{
protected:
    int data1;
public:
    B   `1(int a=8)
```

```
        {     data1 = a;
              cout<<data1<<", Base Constructor1\n";
        }
        ~Base1( )
        {     cout<<data1<<", Base Destructor1\n";}
};
class Base2
{
protected:
       int data2;
public:
        Base2(int a=9)
        {     data2 = a;
              cout<<data2<<", Base Constructor2\n";
        }
        ~Base2( )
        {     cout<<data2<<", Base Destructor2\n";}
};
class Derived:public Base1, public Base2                                    //A
{       int d;
        Base1  b1, b2;                                                      //B
   public:
       Derived(int x, int y, int z) : Base1(x), Base2(y), b1(x+y), b2(x+z)  //C
       {    d=z; cout<<"Derived Constructor\n";}
       ~Derived( )
       {    cout<<"Derived Destructor\n";}
       void Show( )
       {    cout<<data1<<','<<data2<<','<<d<<endl;}
};
void main( )
{    Derived  c(1, 2, 3);
     c.Show( );
}
```

程序的运行结果是:

```
1, Base Constructor1
2, Base Constructor2
3, Base Constructor1          // 构造对象成员b1时的输出
4, Base Constructor1          // 构造对象成员b2时的输出
Derived Constructor
1, 2, 3
Derived Destructor
4, Base Destructor1           // 析构对象成员b2时的输出
3, Base Destructor1           // 析构对象成员b1时的输出
2, Base Destructor2
1, Base Destructor1
```

从运行结果分析，**C++系统处理成：先调用基类的构造函数，再调用对象成员的构造函数，最后调用对象自身的构造函数。**而且调用构造函数的顺序是A、B两行的定义顺序决定的，即使将C行改写成:

```
Derived(int x, int y, int z) : b2(x+z), Base2(y), b1(x+y), Base1(x)  //C
```

本例的输出结果还是一样的。我们还可以看出，析构函数的调用顺序与构造函数相反。

在定义对象和撤销对象时，构造函数和析构函数是必定要调用的，如果我们将C行改写成：

```
Derived(int x, int y, int z)          // C
```

即不显式调用基类和对象成员的构造函数，但系统还是要调用构造函数的，不过此时调用的是缺省构造函数，程序的运行结果会变成：

```
8, Base Constructor1
9, Base Constructor2
8, Base Constructor1
8, Base Constructor1
Derived Constructor
8, 9, 3
Derived Destructor
8, Base Destructor1
8, Base Destructor1
9, Base Destructor2
8, Base Destructor1
```

## 12.5 二义性和支配规则

### 12.5.1 二义性（访问冲突）

在多重继承中，当在派生类中出现两个以上同名的可直接访问的基类成员时，便出现了二义性。有时我们称二义性为访问冲突，如下例所示。

**例12.5** 程序中存在访问二义性的例子，注意本例的继承关系如右图12-6所示。

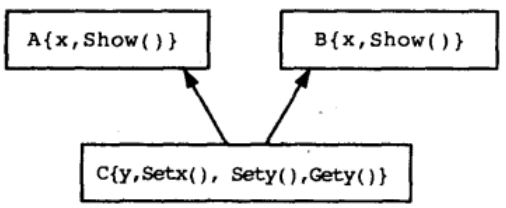



图12-6 多重继承引发二义性

```
#include<iostream.h>
class A
{
protected:
      int  x;
public:
      void Show( ){ cout<<"x="<<x<<'\n';}
      A(int a){ x=a; }
      A( ){ }
};
class B
{
protected:
      int  x;
public:
      void Show( ){ cout<<"x="<<x<<'\n';}
      B(int a){ x=a; }
      B( ){ }
};
class C: public A, public B                    // 公有继承A和B
{
      int  y;
public:
      void Setx(int a) { x=a;}                 // D 类内访问x出现二义性
      void Sety(int b) { y=b;}
```

```
        int Gety( ){ return y; }
};
void main( )
{       C c;
        c.Setx(35);
        c.Sety(300);
        c.Show( );                                    // E  类外访问Show( )出现二义性
}
```

A类和B类的x和Show( )均被继承到C类中，因此，在C类的内部访问x和Show( )时，或在C类的外部访问Show( )时，都存在着二义性问题，如程序中的D行和E行。在D行访问x时，编译器无法判断访问的是哪一个基类中的x。同理，在E行访问Show( )时，编译器无法判断访问的是哪一个基类Show( )。一般地，解决方法有两种，第一种是在定义时，使得不同基类中的各成员，彼此之间不出现同名现象。显然，这不是一个好的解决办法。第二种是，使用作用域运算符来限定所访问成员是属于哪一个基类的。格式为:

<基类类名> :: <成员名>

**例12.6** 解决例12.5的访问二义性。

```
#include<iostream.h>
class A
{
protected:
        int  x;
public:
        void Show( ){ cout<<"x="<<x<<'\n';}
        A(int a){ x=a; }
        A( ){ }
};
class B
{
protected:
        int  x;
public:
        void Show( ){ cout<<"x="<<x<<'\n';}
        B(int a){ x=a; }
        B( ){ }
};
class C: public A, public B                   // 公有继承A和B
{
        int  y;
public:
        void SetAx(int a) { A::x=a;}          // D1   注意这两行如何解决二义性
        void SetBx(int a) { B::x=a;}          // D2
        void Sety(int b) { y=b;}
        int Gety( ){ return y; }
};
void main( )
{       C c;
        c.SetAx(35);
        c.SetBx(45);
        c.Sety(300);
```

```
        cout<<"A::"; c.A::Show( );                    // E1    注意这两行如何解决二义性
        cout<<"B::"; c.B::Show( );                    // E2
        cout<<"y="<<c.Gety( )<<endl;
}
```

程序的运行结果是:

```
A::x=35
B::x=45
y=300
```

当把派生类作为基类，又派生出新的派生类时，这种限定作用域的运算符不能嵌套使用，如下形式的使用方式是不允许的:

```
<类名1> :: <类名2> :: <类名3>...  :: <成员名>
```

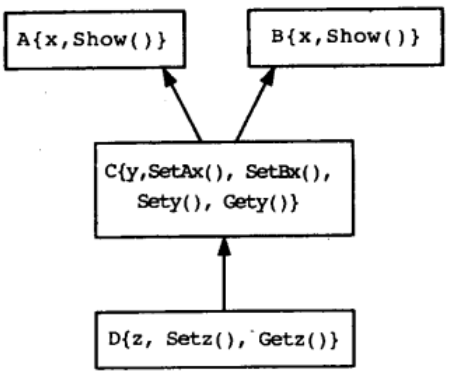



图12-7　多层多重继承中的冲突

**例12.7**　多层继承中的冲突，注意本例的继承关系如图12-7所示。

```
#include <iostream.h>
class  A
{
protected:
       int x;
public:
       void Show( )
       { cout << "x=" << x << '\n' ;  }
};
class  B
{
protected:
       int x;
public:
       void Show( )
       { cout << "x=" << x << '\n' ;  }
};
class  C: public A, public B                          // 公有继承 A类、B类
{
       int y;
public:
       void SetAx(int a) { A::x=a; }
       void SetBx(int a) { B::x=a; }
       void Sety(int b) { y=b; }
       int Gety( ) { return y; }
};
class  D: public C                                    // 公有继承C类
{
       int z;
public:
       void Setz(int a) { z=a; }
       int Getz( ) { return z; }
};
void main(void)
{      D d;
```

```
        d.SetAx(10);  d.SetBx(20);  d.Sety(30);d.Setz(40);
        cout<<"A";
        d.C::A::Show( );                                    //E
        cout<<"B";
        d.C::B::Show( );                                    //F
        cout << "y=" << d.Gety( ) << '\n';
        cout << "z=" << d.Getz( ) << '\n';
}
```

根据前述作用域运算符不能嵌套使用的原则，编译E行和F行时报错。解决的办法可以是，在C类中增加成员函数:

```
void ShowA( ) {cout << "x=" << A::x << '\n'; }
void ShowB( ) {cout << "x=" << B::x << '\n'; }
```

再将E行和F行改写成d.ShowA( ); 和d.ShowB( ); 即可。

当然还可以有一个更简单的解决办法，只要把E行和F行改写成d.A::Show( ); 和d.B::Show( ); 即可。

修改后，程序的运行结果如下:

```
Ax=10
Bx=20
y=30
z=40
```

由于C++是通过作用域运算符来解决访问二义性问题，因此规定任一基类在派生类中只能被直接继承一次，否则会造成成员名访问的冲突。例如:

```
class  A
{
public:
      float  x;
      ...
};
class B: public A, public A    //错误的
{..... };
```

B类的定义是错误的，因为如果直接从A类继承两次，通过作用域运算符就无法解决访问二义性问题。

### 12.5.2 支配规则

在C++中，允许派生类中新增加的成员名与其基类的成员名相同，这种同名并不产生访问二义性(或称访问冲突)。在派生类中访问同名成员时，若没有使用作用域运算符，则访问的是派生类的成员，若使用了作用域运算符，则访问的是基类的同名成员，也就是说，对同名成员的访问，派生类优先，这种优先关系称为支配规则。

**例12.8** 支配规则示例。

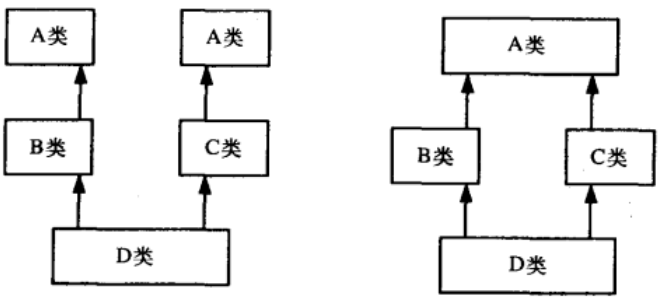
```
#include <iostream.h>
class A
{
protected:
      int x;
public:
      void Set(int a)
```

```
        { x=a; }
        void Show( )
        { cout << "x=" << x << '\n' ; }
};

class B : public A
{
protected:
        int x;
public:
        void SetAx(int a)
        { A::x=a; }                                    //C  访问的是基类A的x
        void SetBx(int a)
        { x=a; }                                       //D  访问的是派生类B的x
        void Show( )
        { cout<<"x="<<x<<endl;  }
};
void main(void)
{       B b;
        b.SetAx(1);
        b.SetBx(2);
        b.A::Show( );                                  //E  访问的是基类A的Show( )
        b.Show( );                                     //F  访问的是派生类B的Show( )
}
```

## 12.6　虚基类

在如图12-8a的公有继承关系中，在D类中包含了基类A的两个拷贝。此时在类D的成员函数中，若欲访问A的成员x，则必须以B::x和C::x区分。




a) 多重继承　　b) 虚基类

图12-8　继承关系

**例12.9**　在派生类中具有两个基类拷贝的例子。

```
#include <iostream.h>
class A
{
protected:
        int x;
public:
        A(int a=0){ x=a; }
};
```

```
class B : public A
{
protected:
      int y;
public:
      B(int a=0, int b=0):A(a)
      { y=b; }
};
class C : public A
{
protected:
      int z;
public:
      C(int a=0, int c=0):A(a)
      { z=c; }
};
class D : public B, public C
{
protected:
      int k;
public:
      D(int a1=0, int a2=0, int b=0, int c=0, int d=0) : B(a1, b), C(a2, c)     //E
      { k=d; }
      void Show( )
      {     cout << "x1=" << B::x << ", ";
            cout << "x2=" << C::x << ", ";
            cout << "y=" << y << ", ";
            cout << "z=" << z << ", ";
            cout << "k=" << k << endl;
      }
};
void main(void)
{     D obj(1, 2, 3, 4, 5);
      obj.Show( );
}
```

注意在E行，调用了基类的构造函数。程序的运行结果是:

```
x1=1, x2=2, y=3, z=4, k=5
```

在多重派生中，若欲使公共的基类在派生类中只有一个拷贝，也就是得到图12-8b的继承关系，则可以将基类A说明成虚基类。在派生类B、C的定义中，只要将基类的类名前加上关键字virtual即可。其一般格式为:

```
class <类名> : virtual <继承方式> <基类名>
{...};
```

或

```
class <类名> : <继承方式> virtual <基类名>
{...};
```

注意画线部分，即“<继承方式>”和“virtual”的前后顺序可随意。

例12.10　虚基类的例子。

```
#include <iostream.h>
class A
{
protected:
      int x;
public:
      A(int a=0)
      { x=a; }
};
class B : virtual public A                                          //E
{
protected:
      int y;
public:
      B(int a=0, int b=0):A(a)
      { y=b; }
};
class C : public virtual A                                          //F
{
protected:
      int z;
public:
      C(int a=0, int c=0):A(a)
      { z=c; }
};
class D : public B, public C
{
protected:
      int k;
public:
      D(int a1=0, int a2=0, int b=0, int c=0, int d=0) : B(a1, b), C(a2, c)  //G
      { k=d; }
      void Show( )
      {    cout << "x=" << x << ", ";                                //H
           cout << "y=" << y << ", ";
           cout << "z=" << z << ", ";
           cout << "k=" << k << endl;
      }
};
void main(void)
{     D obj(1, 1, 2, 3, 4);
      obj.Show( );
}
```

程序的运行结果是:

```
x=0, y=2, z=3, k=4
```

注意在E行、F行虚基类的说明。A类在D的对象中只有一个拷贝，因此在H行对x的访问不会出现二义性。在G行调用了基类的构造函数，由于B类和C类的构造函数B(a1,b)、C(a2, c)分别又调用了A类的构造函数，此时编译器无法确定应该由B类的构造函数还是由C类的构造函数来调用A的构造函数，在这种

情况下，即使a1和a2的值相同，C++仍然约定调用A类的缺省构造函数。所以输出x=0。当然也可以在G行直接调用A类的构造函数，如将G行改为:

```
D(int a1=0, int a2=0, int b=0, int c=0, int d=0) : B(a1, b), C(a2, c), A(a1)   //G
```

则程序的运行结果变为:

```
x=1, y=2, z=3, k=4
```

**例12.11** 虚基类构造函数的调用。

```
#include <iostream.h>
class A
{
public:
   A(char *s = "No Class")
{  cout<<s<<endl; }
   ~A( ){ }
};
class B:public virtual A
{
public:
   B(char *s1, char *s2) : A(s1)                //C
   {    cout<<s2<<endl;}
};
class C:public virtual A
{
public:
   C(char *s1, char *s2) : A(s1)                //D
   {    cout<<s2<<endl;}
};
class D:public B, public C
{
public:
   D(char *s1, char *s2, char *s3, char *s4) : B(s1, s2), C(s1, s3)
   {    cout<<s4<<endl;  }
};
void main( )
{  D d("class A", "class B", "class C", "class D");  }
```

虽然在程序的C行、D行都调用了虚基类的构造函数，而且参数值相等，但系统还是调用了虚基类的缺省构造函数。程序的运行结果如下:

```
No Class
class B
class C
class D
```

## 12.7 访问基类成员和访问对象成员的成员

### 12.7.1 访问对象成员的成员

我们在12.5.1节中已介绍，为了避免访问成员时的二义性，C++规定同一个基类不可以被直接继承两次以上。那么有时在一个类中又必须具有两个已有类的对象，如何来解决这个问题呢？请看下面的例12.12，先定义一个点类Point，由于两点构成一条直线，那么可由两个点的对象构成线类Line。

**例12.12** 访问对象成员的成员。

```
#include <iostream.h>
#include <math.h>
class Point
{
          int x, y;
public:
          Point(int a=0, int b=0){ x=a; y=b;}
          void Setx(int a){x=a;}
          void Sety(int a){y=a;}
          int Getx( ){ return x;}
          int Gety( ){ return y;}
          void Show( )
          {     cout<<"point("<<x<<','<<y<<")\n";}
};
class Line
{
           Point p1, p2;                                          // 对象成员
public:
    Line(int x1, int y1, int x2, int y2): p1(x1, y1), p2(x2, y2) // 调用对象成员构造函数
    { }
    double Length( )
    {    int x1, y1, x2, y2;
         x1=p1.Getx( ); y1=p1.Gety( );                          // 访问对象成员p1的成员
         x2=p2.Getx( ); y2=p2.Gety( );                          // 访问对象成员p2的成员
         return sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
    }
    void Show( )
    {     p1.Show( );                                           // 访问对象成员p1的成员
          p2.Show( );                                           // 访问对象成员p2的成员
          cout<<"Length="<<Length( )<<endl;
    }
};
void main( )
{  Line  line(0, 0, 1, 1);
   line.Show( );
}
```

本程序的运行结果是:

```
point(0, 0)
point(1, 1)
Length=1.41421
```

注意在本程序中，Point类和Line类不是继承关系，只是Point类的两个对象是Line类的对象成员，访问对象成员的成员，与访问一般对象的成员遵循同样的规则。

### 12.7.2 访问基类成员

当然有了Point类后，我们还可以用继承的方法实现Line类，请看下例。

**例12.13** 访问基类成员。

```
#include <iostream.h>
#include <math.h>
```

```
class Point
{
protected:
      int x, y;                    // 定义x、y为保护成员，以使在公有派生类中可直接访问它们
public:
      Point(int a=0, int b=0)     { x=a; y=b;}
      void Setx(int a){x=a;}
      void Sety(int a){y=a;}
      int Getx( ){ return x; }
      int Gety( ){ return y; }
      void Show( )
      {     cout<<"point("<<x<<','<<y<<")\n";  }
};
class Line : public Point                                    // 公有继承
{
protected :
      int x1, y1;
public:
      Line(int a, int b, int c, int d) : Point(a, b)   // 调用基类构造函数
      { x1=c; y1=d; }
      double Length( )
      {  return sqrt((x-x1)*(x-x1)+(y-y1)*(y-y1));}    // A 直接访问基类保护成员x、y
      void Show( )
      {  Point::Show( );                                     // 访问基类成员函数Show( )
         cout<<"point("<<x1<<','<<y1 <<")\n";
         cout<<"Length="<<Length( )<<endl;
      }
};
void main( )
{     Line  line(0, 0, 1, 1);
      line.Show( );
}
```

例12.12和例12.13，在主函数中使用Line类的方式没有什么差别，运行结果也一样。但类的实现是不一样的，本例使用继承的方法实现Line类，在Line类中可以直接访问基类保护成员x、y（见程序A行的注释）。而在例12.12 Line类中是通过对象成员的公有函数接口访问对象成员的成员的。

## 12.8 赋值兼容

在同类对象之间可以相互赋值，那么派生类对象与其基类对象之间能否相互赋值呢？赋值兼容规则为：可以将**公有派生类对象**赋值给基类对象，反之是不允许的。

**例12.14** 赋值兼容规则。

```
#include <iostream.h>
class Point
{
protected:
      int x, y;                                              // 保护成员
public:
      Point(int a=0, int b=0){ x=a; y=b;}
      void Show( )
```

```
        {     cout<<"point("<<x<<','<<y<<")\n";  }
};
class Line : public Point                                    // 公有继承
{
protected:
        int x1, y1;
public:
        Line(int a, int b, int c, int d): Point(a, b)        // 调用基类构造函数
        { x1=c; y1=d; }
};
void main( )
{       Line  line(2, 2, 6, 6);
        Point p;
        p = line;                                            //A
        p.Show( );
}
```

程序运行结果是：Point(2, 2)。

赋值兼容与限制可归结为以下五点：

（1）派生类对象可以赋值给基类对象，系统将派生类对象中从基类继承来的成员赋给基类对象。这一点可以从例12.14的运行结果看出。当然，回忆一下图12-1，可知派生类对象是基类对象的扩充，将派生类对象赋值给基类对象也是合理的。

（2）不能将基类对象赋值给派生类对象，因为派生类对象比基类对象“大”，如果赋值，则派生类对象“大”的部分将无值可赋。

（3）私有或保护继承的派生类对象，不可以赋值给基类对象。

（4）可将派生类对象的地址赋给基类的指针变量。例如 Point *ptr = &line; 。

（5）派生类对象可初始化基类的引用。例如 Point &refp = line; 。

注意，在后两种情况下，使用基类指针或引用时，只能访问从相应基类中继承来的成员，而不允许访问其他基类成员或派生类中增加的成员。

## 习题12

### 一、选择题、填空题

1. 派生类对象可直接访问基类中的_____；在派生类的成员函数中，不可以直接访问的是_____。

A）公有继承的公有成员

B）公有继承的私有成员

C）公有继承的保护成员

D）私有继承的公有成员

2. 关于继承中二义性的描述中_____是错误的。

A）一个派生类的两个基类中有同名成员，在派生类中对这个成员的访问可能出现二义性。

B）解决二义性的最常用的方法是对成员名的限定法。

C）基类和派生类中同时出现的同名函数，也存在二义性问题。

D）派生类D是从基类B和基类C多重派生出来的，而B类和C类又有一个共同的基类A。在派生类D中，访问基类A的成员时也可能出现二义性。

3. 以下说法中错误的是_____。

A）单一继承情况下，派生类对基类的同名成员的访问不会出现二义性。

B）多继承情况下，基类的构造函数的执行顺序取决于定义派生类时所指定的各基类的继承顺序。

C）多继承情况下，基类的构造函数的执行顺序取决于定义派生类构造函数时，在其后所列出的多个基类构造函数的顺序。

D）虚基类是用来解决多继承中公共基类在派生类中只产生一个基类成员拷贝的问题。

4. 设有以下类的定义：

```
#include <iostream.h>
class A
{
protected:
      int a;
public:
      A(int i=0){ a=i; }
      int Geta( ){ return a; }
};
class B: public A
{
      A a1;
public:
      ____________________          //B 类构造函数的定义
      void Show( )
      { cout<<a<<'\t'<<a1.Geta( )<<endl; }
};
```

则关于B类构造函数，下列定义形式一定不正确的是_____。

A）可缺省

B）B(int i=0,int j=0):A(i),a1(j){}

C）B(int i=0,int j=0):A(i),A(j){}

D）B(int i=0,int j=0):A(i){}

5. 设有如下类的定义：

```
class A
{
      int a;
public:
      A(int i=0){ a=i; }
};
class B: public A
{
      int b;
public:
      B(int i=0,int j=0):A(j)
      { b=i; }
};
```

则下列语句组中错误的是 _____。

A）A *pa=new B;

C）B b1(1,2);
   A *pa1=&b1;

B）A a1=B(2,9);

D）A a1(8);
   B b2=a1;

## 二、写出以下程序的运行结果，并完善程序

1.

```
#include <iostream.h>
class A
{
```

```
        int a;
public:
        A( int i=10 ){ a=i; }
        void Show( ){ cout<<"A::a="<<a<<endl; }
};
class B: virtual public A
{
        int b;
public:
        B( int i=20 ):A(8)
        { b=i; }
        void Show( ){ cout<<"B::b="<<b<<endl; }
};
class C: public virtual A
{
        int c;
public:
        C( int i=30 ):A(8)
        { c=i; }
        void Show( ){ cout<<"C::c="<<c<<endl; }
};
class D: public B, public C
{
        int d;
public:
        D( int i=1, int j=2, int k=3 ): B(i),C(j)
        { d=k; }
        void Show( )
        {
                A::Show( );
                B::Show( );
                C::Show( );
                cout<<"D::d="<<d<<endl;
        }
};
void main( )
{       D d1,d2(66,77,99);
        d1.Show( );
        d2.Show( );
}
```

2.

```
#include <iostream.h>
class A
{
protected:
        int x;
public:
        A( int i=10 ){ x=i; }
        void Show( ){ cout<<"A::x="<<x<<endl; }
};
class B: public A
{
```

```
protected:
      int x;
public:
      B( int i=20 )
      { x=i; }
      void Show( ){ cout<<"B::x="<<x<<endl; }
};
class C: public B
{
      int c;
public:
      C( int i=30 )
      { c=i; }
      void Print( )
      {
            cout<<"A::x="<<A::x<<endl;
            cout<<"B::x="<<B::x<<endl;
            Show( );
            cout<<x<<endl;
      }
};
void main( )
{     C c;
      c.Print( );
      c.Show( );
}
```

3. 分析以下程序:

```
#include <iostream.h>
class A
{
      int a;
public:
      A(int i=0)                    //1行
      { a=i; }                      //2行
};
class B
{
      int b;
public:
      B(int i=0)                    //3行
      { b=i; }                      //4行
};
class C: public A
{
      B b1;                         //5行
public:
      void Show( )
      {
          cout<<a<<'\t';            //6行
          cout<<b1.b<<'\n';         //7行
```

```
        }
};
void main( )
{   C c1; c1.Show( );    }
```

程序编译时出现语法错误的行为:

(1) ___行，最简单的修改方法是_____，最合理的修改方法为_____。

(2) ___行，最简单的修改方法是_____，最合理的修改方法为_____。

“最简单”的原则是做最少的改动。“最合理” 的原则是在保护数据成员（即不能通过该类的对象直接访问数据成员）的前提下，做最少的改动。

4. 分析下列程序，根据输出结果完善程序。要求:（1）在主函数中不可以通过对象c1访问类中的所有数据成员。(2) 程序的运行结果为: 3　6　9。

```
#include <iostream.h>
class A
{
(1)_____                                    // 最合理的访问特性
      int a;
public:
      A(int i=0){ a=i; }
};
class B
{
(2)_____                                    // 最合理的访问特性
      int b;
public:
      B(int i=0)
      { b=i; }
};
class C: public A
{
      int c;
      B b1;
public:
      (3) _____                             // 根据运行结果定义构造函数
      {(4)_____}
      void Show( )
      {
      (5) _____                             // 输出c的所有数据成员
      }
};
void main( )
{
      C c1(3,6,9);
      c1.Show( );
}
```

## 三、编程题

要求: 在编程题中，自行设计各类的数据成员和成员函数，尽可能设计得合理一点。

1. 用单一继承方式定义平面坐标体系下的“点”类Point、“线”类Line和“三角形”类Triangle。Line类单一继承Point类，另外增加第2个点坐标。Triangle类单一继承Line类，另外增加第3个点坐标。在主函数中定义一个三角形对象，调用成员函数求出并输出三角形的3个顶点坐标，三角形的周长、面积。将

此题与第10章“习题10”中的编程题第9题做比较。

2. 先定义“高度”类Height和“圆”类Circle，再由Height类和Circle类多重派生出“圆柱体”类Cylinder。在主函数中定义一个圆柱体对象，调用成员函数求出圆柱体的体积和表面积。

3. 定义一个人员类Person，数据成员包括编号（num）、姓名（name）、性别（sex）、家庭住址（faddr）、联系电话（tel）、email地址（email），要求用字符型量实现性别（'f'表示女性，'m'表示男性）成员，其他成员全部用字符数组实现。成员函数包括：① 设置编号，② 设置姓名，③ 设置性别，④ 设置家庭住址，⑤ 设置联系电话，⑥ 设置email地址，⑦ 输出一个人员的全部描述信息。

定义一个描述学生的类Student，该类是由Person类公有派生而来。增加的数据成员包括数学（math）、物理（physics）、外语（eng）、程序设计（prog）四门课的成绩，注意继承来的编号（num）成员在此其意义是学号。注意学号用字符数组实现，各门课的成绩用整数实现。增加的成员函数包括：① 设置学号。② 设置数学、物理、外语、程序设计四门课的成绩，函数原型是void setscore(char tag, int score)，当tag的值分别是'm'、'p'、'e'和'c'时，分别设置数学、物理、外语、程序设计四门课的成绩。③ 输出一个学生的全部描述信息，注意在此函数中调用基类中输出人员的全部描述信息的函数。

定义一个描述教师的类Teacher，该类是由Person类公有派生而来。增加的数据成员包括职务（Headship）、职称（Post）和工资（Salary），注意继承来的编号（num）成员在此其意义是工号。要求用字符数组实现职务和职称，用整型量实现工资。增加的成员函数包括：① 设置职务。② 设置职称。③ 设置工资。④ 输出一个教师的全部描述信息，注意在此函数中调用基类中输出人员的全部描述信息的函数。

在主函数中定义一个学生类对象，再定义一个教师类对象，然后对全体成员函数进行测试。

将此题与第10章“习题10”中的编程题第5题和第6题做比较，体会面向对象编程中“继承”机制的优点。

4. 定义一个日期类Date，再定义一个时间类Time，最后由Date类和Time类多重派生出“日期时间”类DateTime。在主函数完成DateTime类的测试工作。具体要求如下：

- **普通函数（即公用的非成员函数）**：

函数 char *itoa(int n, char *s) 将整数n变换成对应的十进制数的字符串并存放到s所指向的位置。返回该字符串的起始地址。若n的值为356，则s的结果为“356”。此函数供以下两个类成员函数调用。

- **日期类( Date )**：

(1) 私有数据成员：`int Year, Month, Day;` 分别存放年、月、日的值。

(2) 公有成员函数：

构造函数：完成年、月、日的初始化。

函数void SetDate(int, int, int) 重新设置日期。

函数void GetDate(char *) 将整数年、月、日变换成字符串后，以yyyy/mm/dd的形式存放到参数所指向的字符串中。把一个整数变换成字符串由函数itoa( )实现。

- **时间类( Time )**：

(1) 私有数据成员：`int Hour, Minute, Second;` 分别存放小时、分、秒。

(2) 公有成员函数：

构造函数：完成时、分、秒值的初始化。

函数void SetTime(int, int, int) 重新设置时间。

函数void GetTime(char *) 将整数时、分、秒的值变换成字符串后，以hh:mm:ss的形式存放到参数所指向的字符串中。

- **日期和时间类( DateTime )**：

公有成员函数：

构造函数：仅做传递参数用。

函数void SetDateTime(int, int, int, int, int, int) 重新设置日期和时间。

函数void GetDateTime(char *) 将日期和时间变换成字符串后，存放到参数所指向的字符串中。

# 第13章 多 态 性

多态性是面向对象程序设计的重要特性，它与前面讲的封装性和继承性构成面向对象程序设计的三大特性。这三大特性是互相关联的，封装性是基础，继承性是关键，而多态性是补充。

多态分为静态多态和动态多态。函数重载和运算符重载属于静态多态。动态多态是运行时的多态，如某些函数的调用关系在运行阶段才能确定。多态是建立在虚函数基础之上的。

## 13.1 函数重载

所谓函数重载是指赋予一个函数名多个含义。C++中允许在相同的作用域内以相同的名字定义几个具有不同功能的函数，它们可以是普通函数也可以是类的成员函数，这两种情况在前面的章节中都有例子说明。C++编译器在处理对重载函数的调用时，是根据函数的参数类型和参数个数来区分调用哪一个函数的，也就是说多个同名函数，只要它们的参数类型或参数个数不同，它们就是重载函数，编译器就能区分调用哪一个函数。如果有两个函数，它们的参数类型和参数个数完全一样，惟有返回值不同，它们是不能构成重载函数的。

**例13.1** 定义一个字符串string类，对其构造函数进行重载。

```
#include <iostream.h>
#include <string.h>
class String
{
        int Length;
        char *Strp;
public:
        String( );                              // 重载构造函数1
        String(char *s);                        // 重载构造函数2
        String(String &s);                      // 重载构造函数3
        void Print( )
        {       cout<<"String="<<Strp<<endl;
                cout<<"Length="<<Length<<endl;
        }
};
String::String( )                               // 构造函数1：将字符串初始化为空串""
{       Length=0;
        Strp=new char[Length+1];
        *Strp='\0';
}
String::String(char *s)                         // 构造函数2：用s指向的串初始化本类字符串
{       Length=strlen(s);
        Strp=new char[Length+1];
        strcpy(Strp, s);
}
String::String(String &s)                       // 构造函数3：拷贝构造函数
{       Length=s.Length;
        Strp=new char[Length+1];
        strcpy(Strp, s.Strp);
```

```
}
void main( )
{     char *sp="This is a string.";
      String str1;                          // 调用重载构造函数1
      String str2(sp);                      // 调用重载构造函数2
      String str3(str2);                    // 调用重载构造函数3
      str1.Print( );
      str2.Print( );
      str3.Print( );
}
```

程序的执行结果如下:

```
String=
Length=0
String=This is a string.
Length=17
String=This is a string.
Length=17
```

本例重载构造函数，其他成员函数同样可以重载。一般来说重载函数都是功能相类似的函数，提供了使用同样功能的不同形态，增加灵活性。如上例“重载构造函数1”提供初始化空串的方法，“重载构造函数2”提供用字符串指针初始化串的方法，而“重载构造函数3”提供了用已有串初始化新串的方法。

## 13.2 运算符重载

C++中运算符的含义是明确的，但只能用于标准数据类型。对于C++定义的新的数据类型，已有运算符能否使用呢？例如，运算符“+”可以进行整型数据和实型数据的加法运算，如果定义了新的“复数”类型，假定c1和c2是两个复数，能否进行c1+c2运算呢？C++提供的运算符重载机制能够实现将已有运算符应用到新的数据类型变量上，但必须首先进行运算符重载的定义，也就是说给运算符“+”赋予新的含义，使它能适用于新的数据类型。

### 13.2.1 运算符重载的几点说明

#### 1. 可以重载的运算符

在C++中可以重载的运算符有40个，见表13-1。

表13-1 C++中允许重载的运算符

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| + | - | * | / | % | ^ | & | \| |
| ~ | ! | , | = | < | > | <= | >= |
| ++ | -- | << | >> | == | != | && | \|\| |
| += | -= | *= | /= | %= | ^= | &= | \|= |
| <<= | >>= | [ ] | ( ) | -> | ->* | new | delete |

#### 2. 不可以重载的运算符

在C++中不可以重载的运算符有5个，见表13-2。

表13-2 C++中不允许重载的运算符

| | |
|---|---|
| 条件运算符 | ?: |
| 成员运算符 | . |
| 取指向内容运算符 | * |
| 作用域运算符 | :: |
| 求字节数运算符 | sizeof |

#### 3. 重载运算符的限制

(1) 只能对已有运算符重载，不可臆造新的运算符。

(2) 重载运算符时，不允许改变运算符的优先级和结合性。

(3) 重载运算符时，不允许改变运算符的语法结构，如二

元运算符只能重载成二元运算符，一元运算符只能重载成一元运算符。

### 13.2.2 运算符重载的两种方式

运算符重载可以实现将对象作为整体操作，如两个对象之间的加、减法运算等。运算符重载的原理是：一个运算符是一个具有特定意义的符号，只要告诉编译器在什么情况下如何去完成特定的操作，而这种操作是通过特定的函数来实现的。一般地，实现运算符重载有两种形式：成员函数形式和友元函数形式。这两种形式都可以在重载函数中直接访问类的私有成员。我们知道，通过公有函数接口可以间接访问类的私有数据成员，所以在本节的最后，我们介绍第三种实现运算符重载的方法，即通过一般的运算符重载函数（非成员；非友元）实现运算符重载。

#### 1. 重载为类的成员函数

若用成员函数实现运算符的重载，则在类内定义运算符重载函数的格式为：

```
<函数返回值类型> operator <重载运算符>( [<参数列表>] )
{...}
```

在类外定义运算符重载函数的格式为：

```
<函数返回值类型> <类名>::operator <重载运算符> ( [<参数列表>] )
{...}
```

例如，欲重载加法“+”运算符，画线部分可写成 operator + 。我们可以和类的一般成员函数的定义格式比较，operator <重载运算符>相当于一个函数名，选用operator做关键字是为了与普通成员函数区别，使编译器看到operator就知道这是一个运算符重载函数。

**例13.2** 用类的成员函数实现复数类的运算符重载。

```
#include <iostream.h>
class Complex
{
      double Real, Image;
public:
     Complex(double r=0, double i=0)                    //A  构造函数
     { Real=r; Image=i;  }
     Complex operator+(const Complex &c);               //B  重载二元“加”法运算符
     Complex operator+(double);                         //C  重载二元“加”法运算符
     Complex operator-(const Complex &c);               // 重载二元“减”法运算符
     Complex operator-(double);                         // 重载二元“减”法运算符
     Complex operator-(void);                           //D  重载一元“负”号运算符
     Complex operator*(const Complex &c);               // 重载二元“乘”法运算符
     Complex operator/(const Complex &c);               // 重载二元“除”法运算符
     void Show( )
     {     cout<<Real;
           if(Image>0) cout<<'+'<<Image<<'i';            // 如果是正数，应显式输出正号
           else if(Image<0) cout<<Image<<'i';           // 如果是负数，自动输出负号
           cout<<endl;
     }
};
Complex Complex::operator +(const Complex &c)          // 重载二元运算符，“加”法
{
     return Complex(Real+c.Real, Image+c.Image);       // 构造一个临时对象作为返回值
}
Complex Complex::operator+(double r)                   // 重载二元运算符，“加”法
{
     return Complex(Real+r, Image);                    // 构造一个临时对象作为返回值
```

```
}
Complex Complex::operator-(const Complex &c)          //重载“减”法
{    Complex t;
     t.Real=Real-c.Real;
     t.Image=Image-c.Image;
     return t;
}
Complex Complex::operator-(double r)                  //重载“减”法
{    Complex t;
     t.Real=Real-r;
     t.Image=Image;
     return t;
}
Complex Complex::operator-(void)                      //重载一元运算符，“负”号
{    return Complex(-Real, -Image);  }
Complex Complex::operator*(const Complex &c)          //重载“乘”法
{ double r, i;
     r = Real*c.Real - Image*c.Image;
     i = Real*c.Image + Image*c.Real;
     return Complex(r, i);
}
Complex Complex::operator/(const Complex &c)          //重载“除”法
{    double t, r, i;
     t = c.Real*c.Real + c.Image*c.Image;
     r = (Real*c.Real + Image * c.Image)/t;
     i = (Image*c.Real - Real*c.Image)/t;
     return Complex(r, i);
}
void main( )
{     Complex c1(2, 3), c2(4, -2), c3;
      cout<<"c1=";c1.Show( );
      cout<<"c2=";c2.Show( );
      c3=5.0;                                         //E
      cout<<"c3=";c3.Show( );
      c3=c1+c2;                                       //F
      cout<<"c1+c2=";c3.Show( );
      c3=c1+5;                                        //G
      cout<<"c1+5=";c3.Show( );
      c3=c1-c2;
      cout<<"c1-c2=";c3.Show( );
      c3=c1-5;
      cout<<"c1-5=";c3.Show( );
      c3=-c1;                                         //H
      cout<<"-c1=";c3.Show( );
      c3=c1*c2;
      cout<<"c1*c2=";c3.Show( );
      c3=c1/c2;
      cout<<"c1/c2=";c3.Show( );
      c3=(c1+c2)*(c1-c2)*c2/c1;
      cout<<"(c1+c2)*(c1-c2)*c2/c1=";c3.Show( );
}
```

程序的运行结果是:

```
c1=2+3i
c2=4-2i
c3=5
c1+c2=6+1i
c1+5=7+3i
c1-c2=-2+5i
c1-5=-3+3i
-c1=-2-3i
c1*c2=14+8i
c1/c2=0.1+0.8i
(c1+c2)*(c1-c2)*c2/c1=31.8462+25.2308i
```

注意程序中的E行，由于等号两边的数据类型不一致，C++自动处理成c3=Complex(5.0);，调用A行的构造函数完成类型转换（注意第二个参数使用缺省值），完成将其double类型的数据5.0转换成Complex类型的对象，赋值给c3，这就是我们前面讲过的类的构造函数的一种功能。

本质上，运算符的重载就是函数的重载，系统按照约定的函数名来调用重载函数。如程序中F行中的c1+c2;，编译器将c1+c2解释成c1.operator+(c2);，即转化成通过对象c1调用成员函数operator+，参数是c2。编译器将程序中的F行处理成c3=c1.operator+(c2);，调用B行定义的重载函数。同理，编译器将程序中的G行处理成c3=c1.operator+(5);，调用C行定义的重载函数。有了B行和C行定义的两种加法运算符的重载形式，才有F行和G行的两种使用形式。

减法运算与加法运算具有同样的道理。要特别注意，在程序中D行，将减法运算的“-”重载为一元运算符。在程序中H行c3=-c1;，编译器将它处理成c3=c1.operator-();，调用D行定义的重载函数。

我们注意到，用成员函数重载二元运算符时，成员函数有一个参数。二元运算的第一个运算量是对象自身，第二个运算量是成员函数的参数。而用成员函数重载一元运算符时，成员函数没有参数，一元运算的唯一的运算量就是对象自身。

### 2. 重载为友元函数

若用友元函数实现运算符的重载，则在类内定义友元重载函数的格式为:

```
friend <函数返回值类型> operator <重载运算符> ( [<参数列表>] )
{...}
```

由于友元函数不是类的成员函数，因此一般不在类内定义友元函数，在类内只对友元函数做说明。若在类内对友元重载函数已做说明，则在类外定义友元重载函数的格式为:

```
<函数返回值类型> operator <重载运算符> ( [<参数列表>] )
{...}
```

注意友元函数不是类的成员函数，在类外定义时不需要用类名做限定。对二元运算符，友元函数应有两个参数；而对一元运算符，友元函数只有一个参数。

**例13.3**　用友元函数实现复数类的运算符重载。

```
#include <iostream.h>
class Complex
{
        double Real, Image;
public:
        Complex(double r=0, double i=0)                          //A  构造函数
        {  Real=r; Image=i;  }
        friend Complex operator+(const Complex &c1, const Complex &c2);
                                                                 //B  重载二元运算符
        friend Complex operator+(const Complex &c, double r); //C  重载二元运算符
```

```
        friend Complex operator-(const Complex &c1, const Complex &c2);
        friend Complex operator-(const Complex &c, double r);
        friend Complex operator-(const Complex &c);        //D 重载一元运算符
        friend Complex operator*(const Complex &c1, const Complex &c2);
        friend Complex operator/(const Complex &c1, const Complex &c2);
        void Show( )
        { cout<<Real;
          if(Image>0) cout<<'+'<<Image<<'i';              //如果是正数，必须输出正号
          else if(Image<0) cout<<Image<<'i';             //如果是负数，自动输出负号
          cout<<endl;
        }
};
Complex operator +(const Complex &c1, const Complex &c2)
{    return Complex(c1.Real+c2.Real, c1.Image+c2.Image);   }
Complex operator+(const Complex &c, double r)
{    return Complex(c.Real+r, c.Image);  }
Complex operator-(const Complex &c1, const Complex &c2)
{    Complex t;
     t.Real=c1.Real-c2.Real;
     t.Image=c1.Image-c2.Image;
     return t;
}
Complex operator-(const Complex &c, double r)
{    Complex t;
     t.Real=c.Real-r;
     t.Image=c.Image;
     return t;
}
Complex operator-(const Complex &c)
{    return Complex(-c.Real, -c.Image);  }
Complex operator*(const Complex &c1, const Complex &c2)
{    double r, i;
     r = c1.Real*c2.Real - c1.Image*c2.Image;
     i = c1.Real*c2.Image + c1.Image*c2.Real;
     return Complex(r, i);
}
Complex operator/(const Complex &c1, const Complex &c2)
{    double t, r, i;
     t = c2.Real*c2.Real + c2.Image*c2.Image;
     r = (c1.Real*c2.Real + c1.Image * c2.Image)/t;
     i = (c1.Image*c2.Real - c1.Real*c2.Image)/t;
     return Complex(r, i);
}
void main( )
{     Complex c1(2, 3), c2(4, -2), c3;
      cout<<"c1=";c1.Show( );
      cout<<"c2=";c2.Show( );
      c3=5.0;                                          //E
      cout<<"c3=";c3.Show( );
      c3=c1+c2;                                        //F
      cout<<"c1+c2=";c3.Show( );
```

```
    c3=c1+5;                          //G
    cout<<"c1+5=";c3.Show( );
    c3=c1-c2;
    cout<<"c1-c2=";c3.Show( );
    c3=c1-5;
    cout<<"c1-5=";c3.Show( );
    c3=-c1;                           //H
    cout<<"-c1=";c3.Show( );
    c3=c1*c2;
    cout<<"c1*c2=";c3.Show( );
    c3=c1/c2;
    cout<<"c1/c2=";c3.Show( );
    c3=(c1+c2)*(c1-c2)*c2/c1;
    cout<<"(c1+c2)*(c1-c2)*c2/c1=";c3.Show( );
}
```

注意此程序的主函数与例13.2一样，运行结果也一样。

### 3. 两种重载方式的比较

在例13.2和例13.3中，将运算符重载为类的成员函数和友元函数后，在主函数中使用重载运算符的形式上没有什么差别。考虑下面的表达式:

```
c+5.6    //c是Complex类型的对象
```

若重载为类的成员函数，编译器将其处理成 c.operator+(5.6)；若重载为友元函数，编译器将其处理成operator+(c, 5.6)。两种情况在实际执行时都不会出现问题。

如果出现表达式 5.6+c，若重载为友元函数，编译器将其处理成operator+(5.6, c)，由于5.6是double型数据，在函数调用时实参自动转换成形参类型并赋值给形参，相当于operator+(Complex(5.6), c)，此时可以正常处理。如将例13.3中的主函数改为:

```
void main( )
{   Complex c1(2, 3), c2;
    c2=c1+5.6;      c2.Show( );
    c2=5.6+c1;      c2.Show( );
}
```

则运行结果为:

```
7.6+3i
7.6+3i
```

但是若重载成类Complex的成员函数，则编译器将表达式5.6+c处理成 5.6.operator+(c)，显然这种处理无法正确调用Complex类的成员函数。因此，对一般的二元运算符重载为友元函数比重载为成员函数更优越。但是对于赋值运算符，将其重载为成员函数较好，因为赋值运算符是一个二元运算符，其语法格式为 <变量>=<表达式>，第一个运算量必须是对象（变量也是对象），通过对象调用成员函数比较自然。若重载为友元，则可能会出现5.6=c这样的表达式，与赋值表达式的语义不一致。

### 4. 使用非成员、非友元实现运算符的重载

定义友元的目的是在友元函数中直接访问类的私有成员。实际上，可以通过类的公有函数接口访问类的私有成员，所以实现运算符重载，可以使用非成员、非友元实现运算符的重载。但是这种实现方法增加了访问私有数据成员的公有函数的调用时间，不值得提倡。我们可以通过一个实例来体会。

**例13.4**　使用非成员、非友元实现运算符的重载。

```
#include <iostream.h>
class Complex
```

```
{
        double Real, Image;
public:
    Complex(double r=0, double i=0)
    {  Real=r; Image=i;  }
    void SetReal(double Real)
    {  Complex::Real = Real;}
    void SetImage(double Image)
    {  Complex::Image = Image;  }
    double GetReal( ){  return(Real);  }
    double GetImage( ){  return(Image);  }
    void Show( )
    {     cout<<Real;
          if(Image>0) cout<<'+'<<Image<<'i'<<endl;    // 如果是正数，必须输出正号
          else if(Image<0) cout<<Image<<'i'<<endl;    // 如果是负数，自动输出负号
    }
};
Complex operator +(Complex &c1, Complex &c2)          // 二元运算
{                                                     // 函数名为operator +的一个“普
                                                         通”函数
   Complex t;
   t.SetReal( c1.GetReal( )+c2.GetReal( ) );
   t.SetImage( c1.GetImage( )+c2.GetImage( ) );
   return t;
}
void main( )
{   Complex c1(2, 3), c2(4, 8), c3;
    c3=c1+c2;
    c3.Show( );
}
```

程序的运行结果是：6+11i。

### 5. 何时必须重载 = 和 += 运算符？

相同类型的对象之间是可以直接赋值的，C++将赋值处理成两个对象的各个成员直接赋值。如有Complex c1(2, 3),c2;，则将“c2=c1;”处理成“c2.Real=c1.Real; c2.Image=c1. Image;”。一般情况下不会出现问题。但是如果对象中有指针成员指向动态分配的数据空间就会出现问题，下面用两个例子说明。

**例13.5** 在类中，用字符数组实现字符串。

```
#include <iostream.h>
#include<string.h>
class Student
{
      char Num[10];                  // 学号，注意：用数组实现
      char Name[10];                 // 姓名，注意：用数组实现
      int Score;                     // 成绩
public:
      Student(char num[ ]=NULL, char name[ ]=NULL, int score=0)
      {   if(num)strcpy(Num, num);
          else Num[0]='\0';
```

```
            if(name)strcpy(Name, name);
            else Name[0]='\0';
            Score=score;
        }
        void Show( )
        {     cout<<"Num="<<Num<<'\t'
                  <<"Name="<<Name<<'\t'
                  <<"Score="<<Score<<endl;
        }
};
void main( )
{     Student stud1("01201", "Mary", 88), stud2;
      stud2=stud1;                    //A
      stud1.Show( );
      stud2.Show( );
}
```

程序的运行结果是:

```
Num=01201     Name=Mary      Score=88
Num=01201     Name=Mary      Score=88
```

程序中A行使用赋值运算符进行对象整体赋值，C++将其处理成各个成员逐一赋值，即将对象stud2存储空间中的所有内容复制到对象stud1的存储空间中，如图13-1所示。

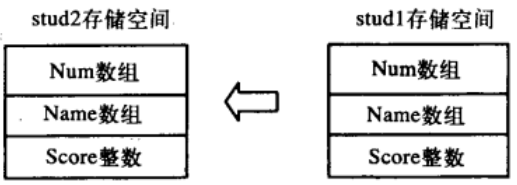



图13-1　同类对象赋值

虽然在类中没有定义赋值运算符的重载函数，但实际上按照C++的默认处理是不会出现问题的。C++默认的处理是: strcpy(stud2.Num, stud1.Num); strcpy(stud2.Name, stud1.Name); stud2.Score = stud1.Score;。

但是如果对象中有指针成员指向动态分配的空间，就会出现问题。

**例13.6**　在类中，用指针实现字符串，即字符串的空间是动态分配的。

```
#include <iostream.h>
#include <string.h>
class Student
{     char *Nump;                          //学号，注意: 用指针实现
      char *Namep;                         //姓名，注意: 用指针实现
      int Score;                           //成绩
public:
      Student(char *nump=NULL, char *namep=NULL, int score=0)
      {   if(nump)
          {     Nump = new char[strlen(nump)+1];
                strcpy(Nump, nump);
          }
          else Nump=NULL;
          if(namep)
          {     Namep = new char[strlen(namep)+1];
                strcpy(Namep, namep);
          }
          else Namep=NULL;
          Score=score;
```

```
        }
        ~Student( )
        {    if(Nump)delete [ ]Nump;
             if(Namep)delete [ ]Namep;
        }
        void Show( )
        {     if(Nump && Namep)
                   cout<<"Num="<<Nump<<'\t'
                       <<"Name="<<Namep<<'\t'
                       <<"Score="<<Score<<endl;
        }
};
void main( )
{     Student stud1("01201", "Mary", 88), stud2;
      stud2=stud1;                              //A
      stud1.Show( );
      stud2.Show( );
      cout.flush( );                            //B
}
```

此程序编译正确，但是运行时，先输出的两行结果和例13.5一样，B行执行结束后，出现错误。B行的作用是将系统输出缓冲区中的内容输出到屏幕上，因为如果没有此行，当程序出现运行错误时，即停止运行，因输出缓冲区未满，在停止运行时，输出缓冲区中的部分内容是不会显示到屏幕上的。按照C++的默认处理，对象赋值是将stud1的成员值逐个赋值给stud2的对应成员，每个对象有两个指针成员，所以赋值后stud2的两个指针也同时指向了stud1的两个指针指向的动态分配的空间（见图13-2）。

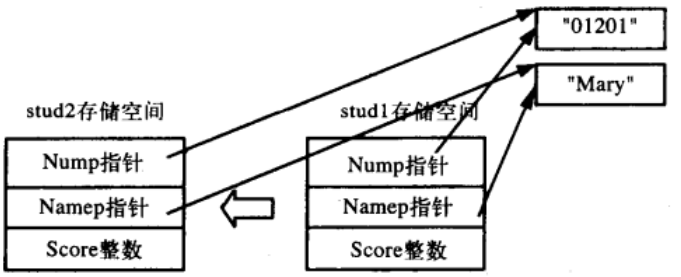



图13-2 含有指针成员的同类对象赋值

程序运行结束时，首先撤销对象stud2，调用析构函数释放stud2的两个指针成员指向的字符串空间。然后撤销对象stud1，又调用析构函数释放stud1的两个指针成员指向的字符串空间，但此时因为这两处空间已经不存在了，所以出现了运行错误。

解决这个问题的方法是为例13.6中的Student类增加赋值运算符重载函数，即类体中增加函数定义如下：

```
Student & operator=(const Student &stud)// 返回值为对象自身的引用
{
     if(Nump)delete [ ]Nump;            //C  释放对象自身的原串空间
     if(Namep)delete [ ]Namep;          //D  释放对象自身的原串空间
     if(stud.Nump)                      // 根据赋值对象的空间大小给被赋值对象分配空间
     {
          Nump = new char[strlen(stud.Nump)+1];
          strcpy(Nump, stud.Nump);
     }
```

```
        else Nump=NULL;
        if(stud.Namep)                          // 根据赋值对象的空间大小给被赋值对象分配空间
        {
              Namep = new char[strlen(stud.Namep)+1];
              strcpy(Namep, stud.Namep);
        }
        else Namep=NULL;
        Score=stud.Score;
        return *this;                           // *this是对象自身
}
```

进行对象赋值，各对象的指针成员指向的空间大小不一定一样，所以在重载赋值运算符时必须先释放被赋值对象的空间，然后再按照赋值对象的空间给被赋值对象分配空间，详见赋值运算符重载函数中的C、D两行。这样，主函数中A行赋值以后，内存的存储状况如图13-3所示。

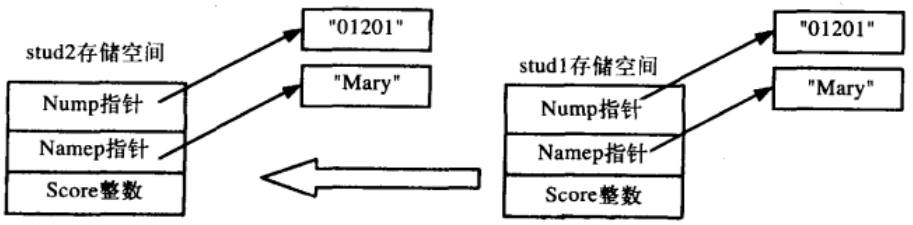



图13-3　重载赋值运算符后，含有指针成员的同类对象赋值

### 6. 对于+=（或=）运算符，重载函数的返回值为void类型或本类类型对象的区别

**例13.7**　+=运算符重载函数的返回值为void。

```
#include <iostream.h>
class Complex
{
        double Real, Image;
public:
        Complex(double r=0, double i=0)
        { Real=r; Image=i;  }
        void operator+=(const Complex &c)          // 返回值为void型
        {
              Real+=c.Real;
              Image+=c.Image;
        }
        void Show( )
        {
              cout<<Real;
              if(Image>0) cout<<'+'<<Image<<'i';    // 如果是正数，必须输出正号
              else if(Image<0) cout<<Image<<'i';    // 如果是负数，自动输出负号
              cout<<endl;
        }
};
void main( )
{
        Complex c1(2, 3), c2(4, -2);
        cout<<"c1=";c1.Show( );
```

```
    cout<<"c2=";c2.Show( );
    c1+=c2;                                          //A
    cout<<"c1+=c2;c1=";c1.Show( );
}
```

运行结果如下:

```
c1=2+3i
c2=4-2i
c1+=c2;c1=6+1i
```

程序中的A行被处理成 c1. operator+=(c2)。如果有连续赋值的表达式，如c1+=c2+=c3，由于等号的运算结合性是从右向左，所以编译器会将上述连续赋值表达式处理成 c1. operator+=( c2. operator+=(c3) )，注意画线部分是第一次赋值，表达式的结果是void类型，此时若欲做第二个赋值就不行了。如果我们将上例中的operator+=( )重载函数改写成:

```
Complex operator+=(const Complex &c) // 返回值为本类类型的对象
{   Real+=c.Real;
    Image+=c.Image;
    return *this;
}
```

此时处理c1+=c2+=c3就没有问题了。

### 7. 对于+=（或=）运算符，返回本类对象与返回本类对象的引用的区别

比较下面两例，先看第一个例子:

```
Complex Complex ::operator+=(const Complex &c)
{   Real+=c.Real;
    Image+=c.Image;
    return *this;
}
```

此函数的返回值为本类对象，C++的处理是：用返回的对象值*this初始化内存临时对象（调用拷贝构造函数），从本函数返回后，用内存临时对象作为调用函数的结果。

再看第二个例子:

```
Complex Complex ::operator+=(const Complex &c)
{   Real+=c.Real;
    Image+=c.Image;
    Complex temp = *this ;
    return temp;
}
```

C++的处理是：调用拷贝构造函数，用返回的对象值temp初始化内存临时对象，内存临时对象作为调用函数的结果。从上面两个小例子可以看出，若重载函数返回对象值，则返回*this和返回temp均可。但要注意，因为返回对象值时需要调用拷贝构造函数初始化内存临时对象，因此若对象有动态分配的存储空间，就必须定义拷贝构造函数。

再看第三个例子:

```
Complex & Complex ::operator+=(const Complex &c)
{   Real+=c.Real;
    Image+=c.Image;
    return *this;
}
```

本例的返回值为本类对象的引用，不需初始化内存临时对象，返回值是对象自身，即*this。因为不

需要初始化内存临时对象，效率较高，所以像 +=、= 等改变对象值的重载函数最好返回对象的引用。

若将第三个例子改写成第四个例子:

```
Complex & Complex ::operator+=(const Complex &c)
{   Real+=c.Real;
    Image += c.Image;
    Complex temp = *this ;
    return temp;
}
```

此时，出现问题。因为返回的是对象的引用（即对象自身，不借助于内存临时对象），系统要求在执行流程返回到调用处时，返回值是存在的。但是本例的返回值是函数内部的局部对象temp，而局部对象在函数返回前，其空间是被撤销的。

**例13.8**　对于字符串类，重载 = 运算符，返回对象自身的引用。就本例而言，可以不定义拷贝构造函数，程序能正确运行。

```
#include <iostream.h>
#include <string.h>
class String
{   char *Strp;
public:
    String(char *s=0)                          // 构造函数
    {    if(s)
         {       Strp=new char[strlen(s)+1];
                 strcpy(Strp, s);
         }
         else Strp=0;
    }
    String & operator = (String & s)           // A   赋值运算符重载函数
    {    if(Strp) delete [ ]Strp;
         if(s.Strp)
         {      Strp=new char[strlen(s.Strp)+1];
                strcpy(Strp, s.Strp);
         }
         else Strp=0;
         return * this;
    }
    ~String( )                                 // 析构函数
    {     if(Strp) delete [ ]Strp;}
    void Show( )
    {     cout<<Strp<<endl;}
};
void main( )
{   String str1("Sample String1");
    String str2;
    str2=str1;                                 // 对象赋值运算
    str2.Show( );                              // 输出 Sample String1
}
```

但是若将程序中A行的函数头部改为: String operator = (String & s)，即返回本类对象，因类中有动态申请的字符串空间，则必须增加定义拷贝构造函数。因为要使用返回的对象去初始化内存临时对象，所以应在类体中增加定义拷贝构造函数如下:

```
String(String & s)  // 拷贝构造函数
{    if(s.Strp)
     {    Strp=new char[strlen(s.Strp)+1];
          strcpy(Strp, s.Strp);
     }
     else Strp=0;
}
```

### 8. 调用拷贝构造函数和调用赋值运算符重载函数的时机

**例13.9** 比较调用拷贝构造函数和调用赋值运算符重载函数的时机。

```
#include <iostream.h>
class Complex
{
          double Real, Image;
public:
   Complex(double r=0, double i=0)           // 构造函数
   {  Real=r; Image=i;  }
   Complex(Complex &c)                       // 拷贝构造函数
   {   Real=c.Real; Image=c.Image;
       cout<<"Call copy "<<Real<<' '<<Image<<"\n";
   }
   Complex operator=(const Complex &c)       // = 运算符重载函数
   {   Real = c.Real;
       Image = c.Image;
       cout<<"Call = "<<Real<<' '<<Image<<"\n";
       return *this;                         // A   初始化内存临时对象
   }
   void Show( )
   {     cout<<Real;
         if(Image>0) cout<<'+'<<Image<<'i'; // 如果是正数，必须输出正号
         else if(Image<0) cout<<Image<<'i'; // 如果是负数，自动输出负号
         cout<<endl;
   }
};
void main( )
{  Complex c1(2, 3), c2(4, -2);
   Complex c3=c1;                            // B 调用拷贝构造函数，此句等价于Complex c3(c1);
   cout<<"Section1"<<endl;
   c1=c2;                                    // C 调用赋值运算符重载函数
   cout<<"Section2"<<endl;
   c1.Show( );
   c3.Show( );
}
```

程序的运行结果如下：

```
Call copy 2 3      // 程序中B行的输出
Section1
Call = 4 -2        // 程序中C行调用 = 重载函数时的输出
Call copy 4 -2     // 程序中A行，用返回值*this初始化内存临时对象时的输出
Section2
4-2i
2+3i
```

在创建新对象时调用拷贝构造函数对新对象进行初始化，如程序中的B行和A行；在B行用c1初始化c3。在A行用*this初始化内存临时对象。在进行对象赋值时，调用的是赋值运算符重载函数，而不是拷贝构造函数，如程序中的C行。

### 13.2.3 类型转换函数——将本类对象转换成其他类对象

C++允许不同数据类型变量之间相互赋值，赋值时系统会自动进行类型转换。

例如：若有Complex c(3, 2); double x=6.2;，则进行赋值 c=x和x=c赋值运算时，C++会处理成c=Complex(x)和x=double(c)，也就是将等号右边的值转换成与等号左边类型一致的值，然后赋给等号左边的变量。对于第一种情况c=Complex(x)，C++调用构造函数完成转换，本例得到的复数c的实部为6.2、虚部为0，请参见例13.2的E行，同时参见10.2.5节。对于第二种情况x=double(c)，应给Complex类定义一个类型转换函数，以完成将本类对象按要求转换成其他类型数据的工作。

在类内定义类型转换函数的一般格式为：

```
operator <目标数据类型> ( )
{ ... }
```

在类外定义类型转换函数的一般格式为：

```
<类名> :: operator <目标数据类型> ( )
{ ... }
```

功能：将<类名>类对象自动转换成<目标数据类型>类对象。注意：(1) 该函数不能有参数。(2) 在函数定义的格式上，不能写返回值类型，因返回值已确定为<目标数据类型>。(3) 类型转换函数只能用成员函数实现，不能用友元函数实现。

**例13.10** 类型转换函数的定义和使用。

```
#include <iostream.h>
class Complex
{
   double Real, Image;
public:
   Complex(double r=0, double i=0)
   {  Real=r; Image=i;  }
   operator double ( )                    //A  类型转换函数，将Complex类转换成double类
   {  return Real; }
};
void main( )
{   Complex c(3,2);
    double x;
    x=c;                                  //B
    cout << "x=" << x << endl;
}
```

系统第一步将B行处理成x = double (c);，进而处理成x = c.operator double ( );。

程序运行，输出x=3。

**例13.11** 成员函数和类型转换函数的比较。

```
#include <iostream.h>
class RMB                                 //定义一个“人民币”类
{
      int Yuan, Jiao, Fen;                //元、角、分
public:
      RMB(int y=0, int j=0, int f=0)
```

```
        {   Yuan=y; Jiao=j; Fen=f;}
        operator int( )                          // 转换成整数值(结果单位: 分)
        {    return (Yuan*100+Jiao*10+Fen);    }
        operator double( )                       // 转换成实数值(结果单位: 元)
        {    return (Yuan+double(Jiao)/10+double(Fen)/100);     }
        int GetFen( )                            // 用成员函数转换成“分”值
        {    return (Yuan*100+Jiao*10+Fen);}
};
void main(void)
{     RMB  r(23, 8, 6);
      int r1, r2, r3;
      r1=r ;                                     // A 处理成 r1 = int(r);, 本质与B行一致
      r2=int(r);                                 // B 处理成 r2 = r.operator int( );
      r3=r.GetFen( );                            // C 调用成员函数得到分值
      cout<<"r1="<<r1<<endl;
      cout<<"r2="<<r2<<endl;
      cout<<"r3="<<r3<<endl;
      double d;
      d=r;                                       // D 处理成 d = r.operator double( );
      cout<<"d="<<d<<endl;
}
```

程序的运行结果如下:

```
r1=2386
r2=2386
r3=2386
d=23.86
```

程序中的B行写出了显式转换的形式，但效果与A行等价。C行调用成员函数实现与类型转换int同样的功能。显然，在这三种方式中，A行最方便。注意D行的转换，转换的结果可以根据需要自己定义。

### 13.2.4 其他运算符的重载

#### 1. 重载++、--运算符

在C++中，一元运算符++ 和--比较特别，它们有前置和后置两种形式。这两个运算符的重载既可以用成员函数又可以用友元函数实现。以“++”运算符为例，用**成员函数**实现重载时，在类外定义前置++运算函数的一般定义格式为:

```
<函数返回值类型> <类名>::operator++ ( )
{ ... }
```

在类外定义后置++运算函数的一般定义格式为:

```
<函数返回值类型> <类名>::operator++ (int)
{ ... }
```

其中int用于区分前后置运算，实际使用时，一般不给实参，当然也可给出实参，但实参没有实际意义。以“++”运算符为例，用**友元函数**实现重载时，在类内定义前置运算函数的一般定义格式为:

```
friend <函数返回值类型> operator++ (<类名> &obj )
{ ... }
```

定义后置运算函数的一般定义格式为:

```
friend <函数返回值类型> operator++ (<类名> &obj, int)
{ ... }
```

其中int用于区分前后置运算。

假定一种货币只有元和角两种单位，一元等于10角，++和--表示加减1角。定义一个类Money 用于实现++和--运算。

**例13.12　实现++和--的前置和后置运算符重载。**

```
#include <iostream.h>
class Money
{
        int Yuan, Jiao;                                  //元、角
public:
        Money(int y=0, int j=0)
        {   Yuan=y; Jiao=j;}
        Money(double d)
        {   Yuan=int(d); Jiao=int((d-Yuan)*10);}
        Money operator++( );                             //①用成员函数实现前置++
        Money operator++(int);                           //②用成员函数实现后置++
        friend Money operator--(Money &m);               //③用友元函数实现前置--
        friend Money operator--(Money &m, int);          //④用友元函数实现后置--
        void Show(char *s)
        {     cout<<s<<':'<<Yuan<<","<<Jiao<<endl; }
};
Money Money::operator++( )                               //①用成员函数实现前置++
{       Jiao++;
        if(Jiao>=10)
        { Yuan+=1; Jiao-=10; }
        return *this;
}
Money Money::operator++(int)                             //②用成员函数实现后置++
{       Money temp=*this;                                //保存对象原始的值
        Jiao++;
        if(Jiao>=10)
        { Yuan+=1; Jiao-=10; }
        return temp;                                     //返回对象原始的值
}
Money operator--(Money &m)                               //③用友元函数实现前置--
{        if(m.Jiao==0)
         {  m.Yuan-=1; m.Jiao=10;  }
         m.Jiao--;
         return m;
}
Money operator--(Money &m, int)                          //④用友元函数实现后置--
{        Money temp=m;                                   //保存对象的原始值
         if(m.Jiao==0)
         {       m.Yuan-=1;                              //借位
                 m.Jiao=10;
         }
         m.Jiao--;
         return temp;                                    //返回对象原始的值
}
void main(void)
{        Money m(15, 3), m1, m2, m3, m4;
```

```
    cout<<"m: "; m.Show("m");
    m1=++m;                                    //处理成: m1 = m.opetator++( );
    cout<<"m1=++m;\n"; m1.Show("m1"); m.Show("m");
    m2=m++;                                    //处理成: m2 = m.opetator++(int);
    cout<<"m2=m++;\n"; m2.Show("m2"); m.Show("m");
    m3=--m;                                    //处理成: m3 = opetator--(m);
    cout<<"m3=--m;\n"; m3.Show("m3"); m.Show("m");
    m4=m--;                                    //处理成: m4 = opetator--(m, int );
    cout<<"m4=m--;\n"; m4.Show("m4"); m.Show("m");
    Money m5(8.5);
    m5.Show("m5");
}
```

程序的运行结果如下:

```
m: m:15, 3
m1=++m;
m1:15, 4
m:15, 4
m2=m++;
m2:15, 4
m:15, 5
m3=--m;
m3:15, 4
m:15, 4
m4=m--;
m4:15, 4
m:15, 3
m5:8, 5
```

在后置运算中，如函数②④，由于函数必须返回对象原始的值，因此用一个临时对象保存对象原始的值，然后再进行对象的++、--运算，最后返回对象的原始值。在函数①③中由于返回值是已被改变了的对象自身的值，为了提高效率可将它们改写为:

```
Money & Money::operator++( )                   //①用成员函数实现前置++
{                                              //返回对象自身的引用
      Jiao++;
      if(Jiao>=10)
      { Yuan+=1; Jiao-=10; }
      return *this;
}
Money & operator--(Money &m)                   //③用友元函数实现前置--
{                                              //返回参数对象的引用
      if(m.Jiao==0)
      {  m.Yuan-=1; m.Jiao=10; }
      m.Jiao--;
      return m;
}
```

注意这两个函数的返回值为对象的引用。为什么②④两个函数的返回值类型不能改写成对象的引用呢？因为返回的是局部对象，局部对象在函数结束后就不存在了。

用成员函数实现前置++，返回值可以是对象也可以是引用，如下所示。这两种实现方法有什么区别呢?

```
Money Money:: operator++()
{       // 返回对象
    Jiao++;
    if(Jiao>=10)
    { Yuan+=1;Jiao-=10;}
    return *this;
}
```

```
Money & Money:: operator++()
{      // 返回引用
     Jiao++;
     if(Jiao>=10)
     {Yuan+=1;Jiao-=10;}
     return *this;
}
```

现在将主函数改写为:

```
void main(void)
{   Money m(15, 3);
    ++(++(++m));
    m.Show("m");
}
```

如果前置++的返回值为对象，则程序的输出为: m:15,4; 。如果前置++的返回值为引用，则程序的输出为: m:15,6。对于表达式 ++(++(++m))，先算内层(++m)，如果返回值为对象，则++m的运算结果是内存临时对象，外层的两次++是针对不同的两个内存临时对象进行的，结果m的值只被加了一次。如果前置++的返回值为引用，则对于表达式 ++(++(++m))，先算内层(++m)，++m的运算结果就是m自身，外层的两次++依然作用在m自身上，结果m的值被加了三次。

### *2. 重载下标运算符

在C++中使用数组时，系统不做下标是否越界的检查，程序中可能会出现一些潜在的错误。学习了面向对象程序设计后，我们可以定义一个功能更强大的数组类，通过重载完成数组下标合法性的检查，另外还可以实现数组整体赋值等功能。

下标运算符重载函数的格式是:

```
<函数返回值类型> <类名>::operator[ ] ( <参数表> )
{ ... }
```

**例13.13** 定义数组类，实现下标越界检查和数组整体赋值。

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
class Array
{
        int Length;                          // 数组元素个数
        int *Arrp;                           // 数组指针
public:
        Array(int size=0);
        ~Array( )
        { if(Arrp) delete [ ]Arrp; }
        int GetLen( )
        { return Length; }
        int & operator[ ](int index);
        void operator=(Array &arr);
};
Array::Array(int size)
{
        Length=size;
        Arrp=NULL;
```

```
        if(Length)
        {     Arrp=new int[Length];
              memset(Arrp, 0, sizeof(int)*Length); // A
        }
}
int & Array::operator[ ](int index)          //返回引用
{       if(index>=Length || index<0)
        {       cout<<"Error: 下标 "<<index<<" 越界\n";
                exit(1);
        }
        return Arrp[index];
}
void Array::operator=(Array &arr)
{    if(Arrp)delete [ ]Arrp;
     Length=arr.Length;
     Arrp=NULL;
     if(Length)
     {     Arrp=new int[Length];
           for(int i=0; i<arr.GetLen( ); i++)
                 Arrp[i]=arr.Arrp[i];
     }
}
void main( )
{      int size;
       cin>>size;                              //输入数组的大小
       Array arr1(size), arr2;
       for(int i=0;i<size;i++)
             cin>>arr1[i];                     //B
       arr2=arr1;                              //数组整体赋值
       for(i=0;i<arr2.GetLen( );i++)
             cout<<arr2[i]<<'\t';              //C
       cout<<endl;
       cout.flush( );
       arr2[5]=3;                              //D
       cout<<arr2[5]<<endl;
}
```

程序第一次的运行结果如下（其中画线部分是用户的输入）：

```
4 <Enter>
1  2  3  4 <Enter>
1       2       3       4
Error: 下标 5 越界
```

程序第二次的运行结果如下（其中画线部分是用户的输入）：

```
8 <Enter>
1  2  3  4  5  6  7  8 <Enter>
1       2       3       4       5       6       7       8
3
```

程序中A行的功能是将内存中从Arrp开始的sizeof(int)*Length个字节清零。在程序的B行出现的arr1[i]，C++会处理成arr1.operator[ ](i)。由于类的成员函数operator[ ]( )返回的是数组第i个元素的引用，所以输入的值被赋给数组第i个元素。同理，在程序的C行出现的arr2[i]，C++会处理成arr2.operator[ ](i)。对程序的

第一次运行，当运行到D行时，由于下标越界，因此输出出错信息。对程序的第二次运行，由于下标没有越界，因此运行不出错。

对于下标运算符的重载，要注意以下几点:

(1) 下标运算符的重载只能通过成员函数来实现，不能用友元实现。

(2) 下标运算符的重载函数只能是非静态的成员函数。

(3) 该函数只能有一个参数。

由于该函数只能有一个参数，因此只能对一维数组重载。对于多维数组，解决办法有两种：一种方法是将多维数组看成一维数组来处理；另一种方法是通过使用**函数调用运算符**来处理。

### *3. 重载函数调用运算符

在C++中，把函数名后的括号( )作为函数调用运算符。函数调用运算符重载函数的格式是:

```
<函数返回值类型> <类名>::operator( )( <参数表> )
{ ... }
```

其中参数个数可以是0个或多个。

**例13.14** 定义矩阵类（相当于二维数组），实现下标越界检查。

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
class Matrix
{
      int Rows, Cols;                          // 二维数组的行数、列数
      int *Mptr;                               // 数组空间起始指针（用一维空间存放二维数组）
public:
      Matrix(int r=0, int c=0)
      {    Rows=Cols=0;
           Mptr=NULL;
           if(r && c)
           {     Rows=r;
                 Cols=c;
                 Mptr=new int[r*c];
                 memset(Mptr, 0, r*c*sizeof(int));
           }
      }
      ~Matrix( )
      {   if(Mptr)delete [ ]Mptr;}
      int & operator ( )(int i, int j)    // 返回引用
      {
          if(i<0 || i>=Rows || j<0 || j>=Cols)
          {      cout<<"下标越界!\n";
                 exit(2);
          }
          return *( Mptr+i*Cols+j );       // A 返回二维数组第i行第j列元素的引用
      }
      int GetRows( )
      { return Rows; }
      int GetCols( )
      { return Cols; }
};
```

```
void main( )
{     Matrix m(4, 4);
      int i, j;
      for(i=1; i<= m.GetRows( ); i++)
            for(j=1; j<= m.GetCols( ); j++)
                  cin>>m(i, j);           // B
      for(i=1; i<= m.GetRows( ); i++)
      {     for(j=1; j<= m.GetCols( ); j++)
                  cout<<m(i, j)<<'\t';
            cout<<endl;
      }
}
```

程序B行中的m(i, j)，C++将其处理成m.operator( )(i, j)，我们注意到该函数返回的是二维数组第i行第j列元素的引用，所以B行输入的数据赋给了数组第i行第j列元素。

程序的运行结果如下（其中画线部分是用户的输入）：

```
1 <Enter>
2 <Enter>
3 <Enter>
下标越界!
```

表示当输入第4个数时，程序的B行执行的是cin>>m(1, 4);，第二维下标越界了，所以出错，退出程序的执行。如果将主函数修改成：

```
void main( )
{  Matrix m(4, 4);
   int i, j;
   for(i=0; i<m.GetRows( ); i++)
         for(j=0; j<m.GetCols( ); j++)
               cin>>m(i, j);
   for(i=0; i< m.GetRows( ); i++)
   {     for(j=0; j<m.GetCols( ); j++)
               cout<<m(i, j)<<'\t';
         cout<<endl;
   }
}
```

则程序的运行结果如下（其中画线部分是用户的输入）：

```
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 <Enter>
1       2       3       4
5       6       7       8
9       10      11      12
13      14      15      16
```

**例13.15** 通过重载函数调用运算符来实现数学函数f(x, y)=(x+5)*y的抽象。

```
#include <iostream.h>
class F
{
public:
      double operator( )(double x, double y)
      { return (x+5)*y; }
};
```

```
void main( )
{
      F f;
      cout<<"f(1.5, 2.2)="<<f(1.5, 2.2)<<endl;
      cout<<"f(1.2, 9.3)="<<f(1.2, 9.3)<<endl;
}
```

程序的运行结果如下:

```
f(1.5, 2.2)=14.3
f(1.2, 9.3)=57.66
```

### *4. 重载new和delete运算符

new和delete运算符用于动态申请和释放存储空间，非常有用。如果没有什么特殊需要，使用系统定义的new和delete运算符就可以动态申请和释放用户定义的新类型的对象空间。当用户使用new和delete运算符时，系统自动调用新类型的构造函数和析构函数。

**例13.16** 使用系统的new和delete运算符解决对象空间的动态分配和释放。

```
#include <iostream.h>
#include <string.h>
class String
{
    int Length;
    char *Strp;                                  // 指向串的指针
public:
    String( );                                   // 重载构造函数1
    String(char *s);                             // 重载构造函数2
    String(String &s);                           // 重载构造函数3
    void Print( )
    {     cout<<"String="<<Strp<<endl;
          cout<<"Length="<<Length<<endl;
    }
    ~String( )                                   // 析构函数
    {     if(Strp) delete[ ]Strp;
          cout<<"Destructor\n";
    }
};
String::String( )
{  Length=0;
    Strp=new char[Length+1];
    *Strp='\0';
    cout<<"Constructor 1\n";
}
String::String(char *s)
{   Length=strlen(s);
    Strp=new char[Length+1];
    strcpy(Strp, s);
    cout<<"Constructor 2\n";
}
String::String(String &s)
{   Length=s.Length;
    Strp=new char[Length+1];
```

```
    strcpy(Strp, s.Strp);
    cout<<"Constructor 3\n";
}
void main( )
{   String *ptr1, *ptr2;
    ptr1 = new String("Student");              // 调用重载构造函数2
    ptr2 = new String(*ptr1);                  // 调用重载构造函数3
    ptr1->Print( );
    ptr2->Print( );
    delete ptr1;
    delete ptr2;
}
```

程序的运行结果是:

```
Constructor 2
Constructor 3
String=Student
Length=7
String=Student
Length=7
Destructor
Destructor
```

如果需要重载new和delete运算符。则重载new运算符的格式为:

```
void * <类名>::operator new(size_t size, [<参数表>] )
{ ... }
```

其中返回值是一个任意类型的指针void *，第一个参数必须是size_t类型。在C++中size_t定义为:

```
typedef unsigned int size_t;
```

size_t是数据类型名，它与unsigned int等价。在头文件stddef.h 、mem.h、alloc.h、stdlib.h中均有这种类型的定义，在重载new运算符时，可以包含这几个头文件中的任意一个。因此在重载new运算符的格式中形参size是一个无符号整型变量，它表示要求分配的存储空间的字节数。实际上这个参数是系统自动产生的，它的值是调用new函数的对象的长度。<参数表>按实际需要给定，可有可无。通常，new重载后，其功能仍然是分配size个字节的存储空间，并返回该空间的起始地址。当然，还可以根据需要做一些其他的操作。

重载delete运算符的格式为:

```
void <类名>::operator delete(void *p, [ size_t size ] )
{ ... }
```

该函数没有返回值。它至少要有一个参数p，是一个任意类型的指针，通常是new运算符得到的指针；第二个参数是任选的，如果有，它的类型必须是size_t。一般来说，重载delete函数，完成的主要功能仍然是释放p指向的空间。

关于new和delete运算符的重载，有以下三点说明:

(1) 必须通过成员函数来实现重载，不能用友元实现重载。

(2) 总是重载为静态的成员函数。不管是否使用了关键字static，编译程序总是将这两个重载函数看成是静态成员函数。

(3) 不能是虚函数。

**例13.17** 重载new和delete运算符。

```
#include <iostream.h>
#include <string.h>
```

```
class String
{
    int Length;
    char *Strp;
public:
    void * operator new(size_t size);                    // 重载运算符new    函数1
    void * operator new(size_t size,char *str);          // 重载运算符new    函数2
    void * operator new(size_t size,String &s);          // 重载运算符new    函数3
    void Print( )
    {     cout<<"String="<<Strp<<endl;
          cout<<"Length="<<Length<<endl;
    }
    void operator delete(void *p)                        // A重载 delete
    {     if(p)
          {    String *t=(String *)p;
               if(t->Strp) delete [ ](t->Strp);
               delete [ ] (char *) p;
          }
         cout<<"Delete\n";
    }
};
void * String::operator new(size_t size)                 // B
{   String *p=(String *)new char[size];
    p->Length=0;
    p->Strp=new char[p->Length+1];
    *(p->Strp)='\0';
    return (void *)p;
    cout<<"New 1\n";
}
void * String::operator new(size_t size, char *str)      // C
{   String *p=(String *)new char[size];
    p->Length=strlen(str);
    p->Strp=new char[p->Length+1];
    strcpy(p->Strp, str);
    cout<<"New 2\n";
    return (void *)p;
}
void * String::operator new(size_t size, String &s)      // D
{   String *p=(String *)new char[size];
    p->Length=s.Length;
    p->Strp=new char[p->Length+1];
    strcpy(p->Strp, s.Strp);
    cout<<"New 3\n";
    return (void *)p;
}
void main( )
{   String *ptr1, *ptr2;
    ptr1 = new ("Student")String;                        // E
    ptr2 = new (*ptr1)String;                            // F
    ptr1->Print( );
```

```
    ptr2->Print( );
    delete ptr1;                                    //G
    delete ptr2;
}
```

程序的运行结果为:

```
New 2
New 3
String=Student
Length=7
String=Student
Length=7
Delete
Delete
```

由于new和delete运算符总是重载为静态的成员函数，而静态的成员函数是属于类的，因此程序中E行的new ("Student")String; 被处理成String::operator new(sizeof(String),"Student")，调用C行的函数，将返回值赋值给ptr1。程序中F行的new (*ptr1)String;被处理成String::operator new((*ptr1), sizeof(String))，调用D行的函数，将返回值赋值给ptr2。程序中G行的delete ptr1;被处理成String::operator delete((void *)ptr1)，调用A行的函数。

### 13.2.5 字符串类

前面介绍的运算符重载的例子较简单，现在定义一个字符串类作为重载运算符的综合应用。在C++中，系统提供的字符串处理能力较弱，如不能直接对字符串对象进行加、减等操作。用本节定义的字符串类，通过运算符重载，就可以对字符串直接操作，使得对字符串的操作就像对一般数据的操作一样方便。

下面定义字符串类String，在类中重载“=”运算符，以实现字符串的直接赋值；重载“+”运算符，以实现两个字符串的拼接；重载“-”运算符，以实现从一个字符串中删除子串的功能；重载“>”、“<”、“==”运算符，以实现字符串的直接比较；同时也定义了转换函数，以将String对象赋值给一般的字符串指针，还定义了一些其他函数，详见下面的说明。

在本例中，String类的完整定义放在头文件“Li1318.h”中，对类进行测试的主函数放在源文件“Li1318.cpp”中。

**例13.18** 定义字符串类String，并测试重载的运算符以及成员函数。

```
//Li1318.h
#include <iostream.h>
#include <string.h>
class String
{
protected:
        int Length;                                 //字符串的长度
        char *Strp;                                 //指向字符串的指针
public:
        String( ){ Strp=NULL; Length=0; }           //缺省构造函数
        String(const char *s);                      //构造函数，以字符指针作为参数
        String(const String &);                     //拷贝构造函数，以字符串对象的引用作为
                                                      参数
        ~String( )
        { if(Strp) delete [ ] Strp; }
        const char *IsIn( const char ) const;       //返回参数字符在字符串中第一次出现的地址
        int IsSubStr(const char *) const;           //判断参数字符串是否为子串
```

```
        void Show( )                                    // 输出字符串
        { cout << Strp << '\n'; }
        int GetLen( ) { return Length; }                // 返回字符串的长度
        const char * GetString( ) { return Strp; }      // 取字符串首指针
        operator const char* ( ) const                  // A 类型转换函数，返回指向常量的字符串指针
        { return (const char *) Strp; }
        String & operator = (String &);                 // 重载赋值运算符
        friend String operator+(const String &, const String & );  // 友元实现+重载
        friend String operator-(const String &, const char * );    // 友元实现－重载
        int operator < ( const String &) const;
        int operator > ( const String &) const;
        int operator == ( const String &) const;
};
String::String(const char *s)                           // 构造函数，以字符指针作为参数
{       if(s)
        {       Length = strlen(s);
                Strp = new char[Length+1];
                strcpy(Strp, s);
        }
        else
        { Strp = NULL; Length = 0; }
}
String::String(const String &s)                         // 拷贝构造函数，以字符串对象的引用作为参数
{       Length = s.Length;
        if(s.Strp)
        {       Strp = new char[Length+1];
                strcpy(Strp, s.Strp);
        }
        else Strp = NULL;
}
const char *String::IsIn( const char c) const          // B  判断字符c是否在字符串中，
                                                        // 如果在，返回其第一次出现时的地址，否
                                                           则返回空指针NULL
{       char *p=Strp;
        while(*p)
          if(*p++ == c) return --p;
        return NULL;
}
int String::IsSubStr(const char *s) const               // 判断s所指向的字符串是否为类中
{                                                       // 字符串的子串。若是，返回1；若不是，
                                                           返回0。
        if(strstr(Strp, s)) return 1;                   // C  strstr( ) 为字符串处理库函数
        else return 0;
}
String & String::operator = (String &s)                // 实现赋值运算
{
        if(Strp) delete [ ] Strp;                       // 释放对象自身的空间
        Length = s.Length;
        if(s.Strp)
        {       Strp = new char[Length+1];
                strcpy(Strp, s.Strp);
```

```
        }
        else Strp = NULL;
        return *this;
}
String operator+(const String &s1, const String &s2 )// 拼接两个字符串
{       String t;
        t.Length = s1.Length + s2.Length;
        t.Strp = new char [ t.Length + 1 ];
        strcpy(t.Strp, s1.Strp);
        strcat(t.Strp, s2.Strp);
        return t;
}
String operator-(const String &s1, const char *s2 ) // D 删除s1中第一次出现的s2
{       String t;
        char *p1=s1.Strp, *p2;
        int i=0, len=strlen(s2);
        if(p2=strstr(s1.Strp, s2))                    // 如果是子串
        {       t.Length=s1.Length - len;
                t.Strp = new char[t.Length+1];
                while(p1<p2)
                        t.Strp[i++]=*p1++;
                p1+=len;
                while(t.Strp[i++]=*p1++);
        return t;
        }
        else
                return s1 ;
}
int String::operator < ( const String &s) const  // 实现小于比较，成员函数实现
{ return(strcmp(Strp, s.Strp)<0 ); }
int String::operator > ( const String &s) const  // 实现大于比较，成员函数实现
{ return(strcmp(Strp, s.Strp)>0 ); }
int String::operator == ( const String &s) const // 实现恒等于比较，成员函数实现
{ return(strcmp(Strp, s.Strp)==0 ); }
//Li1318.cpp
#include "Li1318.h"
void Test_IsIn(char c, String &s)
{
      cout<<"\'"<<c<<"\' in "; s.Show( );
      if(s.IsIn(c))                                   // 测试类的成员函数IsIn( )
      {                                               // s.IsIn(c)返回的是c第一次在s中出现
                                                        的地址
            cout <<"Yes:"<<s.IsIn(c)<<endl;
      }
      else cout <<"No:\n";
}
void Test_IsSubStr(char *sp, String &s)
{     cout<<"\""<<sp<<"\" in "; s.Show( );
      if(s.IsSubStr(sp))                              // 测试类的成员函数IsSubStr( )
            cout <<"Yes:\n";
      else cout <<"No:\n";
```

```
}
void main(void)
{     String s1("C++程序设计 "), s2, s3("学生学习 ");
      String s, s5;
      s1.Show( );
      s2=s1;                                          //测试赋值运算符“=”
      s2.Show( );
      s=s3+s2;                                        //测试字符串拼接运算符“+”以及赋值运
                                                        算符“=”
      s.Show( );
      s5=s-s1;                                        //测试字符串相减运算符“-”
      s5.Show( );
      String s6="C++ programming! ";
      Test_IsIn('g', s6);                             //判断一个字符是否在“串”中
      Test_IsIn('k', s6);
      Test_IsSubStr("prog", s6);                      //判断字符串是否是“串”的子串
      Test_IsSubStr("red", s6);
      cout<<"s1=";s1.Show( );
      cout<<"s3=";s3.Show( );
      if(s1>s3)                                       //字符串比较
            cout<<"s1>s3 OK!\n";
      else if(s1<s3)                                  //字符串比较
            cout<<"s1<s3 OK!\n";
      else if(s1==s3)                                 //字符串比较
            cout<<"s1==s3 OK!\n";
      const char *sp1, *sp3;
      sp1=s1; sp3=s3;                                 //调用A行的类型转换函数，由String类
                                                        转换成const char *类
      cout<<sp3<<sp1<<endl;
      cout<<s3.GetString( )<<endl;
}
```

程序的运行结果如下：

```
C++程序设计
C++程序设计
学生学习 C++程序设计
学生学习
'g' in C++ programming!
Yes:gramming!
'k' in C++ programming!
No:
"prog" in C++ programming!
Yes:
"red" in C++ programming!
No:
s1=C++程序设计
s3=学生学习
s1<s3 OK!
学生学习 C++程序设计
学生学习
```

类String的一些成员函数的参数列表圆括号后有一个const，如程序中A行：

```
const char *String::IsIn( const char c) const
```

画线部分的const表示：（1）在该函数中只能使用数据成员，不能修改数据成员的值。（2）该函数只能调用其他const函数。 IsIn( )函数的功能是判断字符c是否包含在对象的字符串中，如果包含，则返回其第一次出现时的地址，否则返回空指针NULL。注意：返回的是const char *类型的指针，即指向常量的指针。也就是说，不能通过该指针修改它指向的字符串内容。

在C行用到了一个系统标准字符串操作函数strstr( )，该函数原型在头文件string.h中的说明为：char *strstr(const char * s1, const char * s2);，其功能是，若s2是s1的子串，则返回s2在s1中第一次出现时的起始位置（起始地址）；若不是子串，则返回空指针NULL。

D行中的String operator-(const String &s1, const char *s2 )函数，重载了“-”运算符，其功能是：删除s1中第一次出现的子串s2，返回删除子串后的结果字符串；如果s1中不出现s2，则原样返回s1。

**例13.19** 从字符串类String派生出新字符串类Str，说明重载运算符的继承性。

```
//Li1319.cpp
#include "Li1318.h"
class Str : public String
{
public:
    Str( ):String( ){ }                          // 调用基类构造函数
    Str(char *ps):String(ps){ }                  // 调用基类构造函数
    Str(Str &s):String(s){ }                     // 调用基类构造函数
    Str operator+(Str &s)                        // A
    {     Str t;
          t.Length = Length + s.Length;
          t.Strp = new char [ t.Length + 1 ];
          strcpy(t.Strp, Strp);
          strcat(t.Strp, s.Strp);
          return t;
    }
    Str operator=(Str &s)                        // B
    {
          if(Strp) delete [ ] Strp;
          Length = s.Length;
          if(s.Strp)
          {      Strp = new char[Length+1];
                 strcpy(Strp, s.Strp);
          }
          else Strp = 0;
          return *this;
    }
};
void main(void)
{   Str s1("学生学习 "), s2("C++程序设计 ");
    Str s3(s2), s4,s5;
    cout << "s1=";s1.Show( );
    cout << "s2=" << s2 << endl;                 // C   调用继承来的“类型转换”函数
    cout << "s3=" << s3 << endl;
    s4=s2;                                       // 字符串整体赋值，使用重新定义的重载函数
    cout << "s4=s2=" << s4 << endl;
```

```
    s5=s1+s2;                                    // 字符串拼接，使用重新定义的重载函数
    cout << "s5=s1+s2;s5=" << s5 << endl;
    cout << "p=" << s2.IsIn('+') << endl;        // 使用继承来的成员函数
    if(s2>s3)                                    // 使用继承来的字符串比较运算符>
        cout<<"s2>s3 OK!\n";
    else if(s2<s3)                               // 使用继承来的字符串比较运算符<
        cout<<"s2<s3 OK!\n";
    else if(s2==s3)                              // 使用继承来的字符串比较运算符==
        cout<<"s2==s3 OK!\n";
}
```

程序的运行结果是:

```
s1=学生学习
s2=C++程序设计
s3=C++程序设计
s4=s2=C++程序设计
s5=s1+s2;s5=学生学习 C++程序设计
p=++程序设计
s2==s3 OK!
```

Str类公有继承String类，没有增加新的数据成员。虽然基类中定义了运算符“+”的重载函数，但是是用友元函数实现的。友元函数不是类的成员函数，所以在派生类中不被继承，故此在A行重新定义了“+”的重载函数。同理，由于Str类中没有重新定义“-”的重载函数，因此在Str类中不可使用运算符“-”。

虽然在基类中用成员函数重载了“=”运算符，但派生类是不能继承“=”运算符重载函数的，所以在Str类中（程序的B行），重新定义了“=”的重载函数。

赋值运算符operator=( )不能被派生类继承的原因是，派生类的赋值除了要复制基类的成员外，还应对新增加的成员进行复制。如果允许继承“=”运算符，则只能复制基类中的成员。由于派生类的赋值语义与基类的赋值语义不一样，因此派生类不能继承基类中的赋值运算符重载函数operator=( )。另外，成员函数operator=( )也不能被定义为虚函数，见13.4.1节。

注意程序中的C行:

```
cout << "s2=" << s2 << endl;
```

编译器会把C行处理成:

```
cout << "s2=" << (const char *)s2 << endl;
```

为什么呢？在第14章我们会介绍，cout是标准输出流类的对象。读者可以在头文件ostream.h中看到，在ostream类中对运算符“<<”做了重载，使得可以输出C++预先给定的任意数据类型。例如: `int x=5; cout<<x;`，编译器会把`cout<<x;`处理成cout.operator <<(x)，调用输出流类的成员函数operator <<输出x。但是对用户可能定义的新类，C++并没有预先定义运算符重载函数，使ostream类的成员函数operator<<函数可以输出用户定义的Str类型的数据。在第14章我们会告诉大家如何定义运算符重载函数，实现输出用户定义的新类型的数据。对本例中的C行`cout<<"s2="<<s2<<endl;`，编译器是如何处理的呢？

注意以下原则: 当对象（如cout）参与运算时，编译器首先查看该运算符是否是对象所属类的成员函数实现的运算符: (1) 若是，则调用成员函数来实现运算。(2) 若不是，编译器试图将友元函数作为重载的运算符。(3) 若再不是，编译器尝试将参与运算的第2个对象（如s2）转换成本类能处理的类型进行运算。(4) 若仍不成功，则编译器给出出错信息。

程序中的C行就是编译器判断到第（3）步时，用Str类中定义的类型转换函数（Str类从String类继承了类型转换函数，见例13.18中的头文件Li1318.h中的A行）将s2转换成(const char *)s2后，再调用输出流重载函数处理的。如果将该类型转换函数删去，则编译器对例13.19中的源程序文件Li1319.cpp的C行报错:

“error C2679: binary '<<' : no operator defined which takes a right-hand operand of type 'class Str' (or there is no acceptable conversion)”。

## 13.3 静态联编

联编是指一个计算机程序彼此关联的过程，在本章中指**函数间调用关系的确定**。按照联编所确定的时刻不同，可分为两种：静态联编和动态联编。

静态联编是指联编出现在编译连接阶段，这种联编又称早期联编，通过这种联编可实现静态多态，例如前面介绍的函数重载。运算符重载是通过函数重载实现的，所以在这里我们统称函数重载。

**例13.20** 普通函数的静态联编。

```
#include <iostream.h>
int add(int a, int b)                          // 重载函数1
{ return(a+b); }
double add(double a, double b)                 // 重载函数2
{ return(a+b); }
void main( )
{
    cout<<add(1, 2)<<'\t';                     // 编译时确定调用重载函数1
    cout<<add(1.1, 2.2)<<'\n';                 // 编译时确定调用重载函数2
}
```

此例，在编译连接阶段，就能根据参数的个数和类型确定调用的是哪一个函数。

**例13.21** 类的成员函数的静态联编。

```
#include <iostream.h>
class Point
{
protected:
    double x, y;                    // 点的坐标值
public:
    Point(double a=0, double b=0)
    { x=a; y=b; }
    double Area( )                  // 函数1
    { return 0.0; }
};
class Rectangle : public Point
{
protected:
    double x1, y1;                  // 长方形右下角点的坐标值，基类中x，y为左上角坐标点
public:
    Rectangle(double a=0, double b=0, double c=0, double d=0):Point(a, b)
    { x1=c; y1=d; }
    double Area( )                  // 函数2
    { return (x-x1)*(y-y1); }
};
class Circle:public Point
{
protected:
    double r;                       // 半径，基类中x，y为圆心坐标点
public:
    Circle(double a=0, double b=0, double c=0):Point(a, b)
```

```
    { r=c; }
    double Area( )                    // 函数3
    { return 3.14*r*r; }
};
double CalcArea(Point &p)
{
    return(p.Area( ));                // A  编译连接时确定调用函数1
}
void main( )
{
    Rectangle r(0, 0, 1, 1);
    Circle c(0, 0, 1);
    cout<<CalcArea(r)<<'\t';
    cout<<CalcArea(c)<<'\n';
}
```

程序中A行的p.Area( )在编译连接时确定调用函数1，因为p是Point类对象，而Area( )是其成员函数。这是通过静态联编实现的。程序的运行结果是:

```
0   0
```

能否找到一种机制，让CalcArea( )函数变成一个通用的求面积的函数，也就是说在调用CalcArea( )函数时，根据实参对象的不同而求出不同形状的面积呢？即CalcArea(r)求长方形r的面积，而CalcArea(c)求圆形c的面积。这就是C++提供的动态联编和虚函数完成的工作。

## 13.4　动态联编和虚函数

程序中若出现函数调用，但在程序的编译连接阶段无法确定调用哪一个函数，而只有到了程序的运行阶段才能确定调用的是哪一个函数，这就是动态联编。动态联编又称滞后联编、晚期联编，动态联编技术能实现动态多态。C++中提供动态联编技术，它必须将类的成员函数定义成虚函数，才可以实现动态联编。

### 13.4.1　虚函数的定义和使用

在类内，将成员函数定义成虚函数的格式为:

```
virtual <函数返回值类型> <函数名>( [<参数列表>] )
{...}
```

**例13.22**　将成员函数定义成虚函数，以实现动态联编。

```
#include <iostream.h>
class Point
{
protected:
    double x, y;                                  // 点的坐标值
public:
    Point(double a=0, double b=0)
    { x=a; y=b; }
    virtual double Area( )                        // 虚函数1
    { return 0.0; }
};
class Rectangle:public Point
{
protected:
```

```
    double x1, y1;                                  // 长方形右下角点的坐标值，基类中x, y为
                                                       左上角坐标点
public:
    Rectangle(double a=0, double b=0, double c=0, double d=0):Point(a,b)
    { x1=c; y1=d; }
    virtual double Area( )                          // 虚函数2
    { return (x-x1)*(y-y1); }
};
class Circle:public Point
{ protected:
    double r;                                       // 半径，基类中x, y为圆心坐标点
public:
    Circle(double a=0, double b=0, double c=0):Point(a, b)
    { r=c; }
    virtual double Area( )                          // 虚函数3
    { return 3.14*r*r; }
};
double CalcArea(Point &p)
{  return(p.Area( ));   }                           // A
void main( )
{   Point p(1, 2);
    Rectangle r(0, 0, 1, 1);
    Circle c(0, 0, 1);
    cout<<CalcArea(p)<<'\t'<<CalcArea(r)<<'\t'<<CalcArea(c)<<'\n';
}
```

程序的运行结果是:

```
0       1       3.14
```

注意程序中三个画线的函数，它们都被定义成了虚函数。这时C++规定对A行p.Area( )函数调用的处理方法是：在编译连接阶段，已知A行调用的Area( )是虚函数，那么编译连接程序不确定调用哪一个函数，而是在此处保留基类Point和其派生类Rectangle、Circle相关的三个虚函数Area( )的三个入口地址，在程序的运行阶段，根据实参的类型来确定调用“虚函数1”、“虚函数2”或“虚函数3”。例如：如果实参是Rectangle类对象，那么在A行调用的是“虚函数2”，依此类推。

关于虚函数的使用，有以下几点说明:

(1) 当基类中把成员函数定义为虚函数后，在其派生类中的虚函数必须与基类中的虚函数同名，且函数的参数个数、参数类型必须完全一致，否则属于函数的重载，而不是能实现动态多态的虚函数。

(2) 基类中虚函数前的关键字virtual不能缺省，派生类中虚函数前的关键字virtual可以缺省，缺省后仍然是虚函数。

(3) 动态多态必须通过**基类对象的引用**或**基类对象的指针**调用虚函数才能实现。例如可以将例13.22改写成如下形式，运行结果不变:

```
......                          // 类的定义部分一样，在此省略
double CalcArea(Point *p)       // 形参是基类对象的指针
{
   return(p->Area( ));          // 通过指针调用成员函数
}
void main( )
{
   Point p(1, 2);  Rectangle r(0, 0, 1, 1);   Circle c(0, 0, 1);
   cout<<CalcArea(&p)<<'\t'<<CalcArea(&r)<<'\t'<<CalcArea(&c)<<'\n';   // 实参是指针
}
```

（4）虚函数必须是类的成员函数，不能是友元函数，也不能是静态成员函数。

（5）不能将构造函数定义为虚函数，但可将析构函数定义为虚函数。如果类的构造函数中有动态申请的存储空间，则在析构函数中释放该空间，此时，建议将析构函数定义为虚函数，以便实现撤销对象时的多态性。

（6）虚函数与一般函数相比，调用时的执行速度要慢一些。因为，为了实现动态联编，在每个派生类中都要保存相应虚函数的入口地址，而且函数的调用机制也是间接实现的。因此，除非要编写一些通用函数并且必须使用虚函数才能完成其功能外，一般不要使用虚函数。

**例13.23** 成员函数调用虚函数。

```
#include <iostream.h>
class A
{
public:
    virtual void fun1( )                          //D  虚函数
    { cout << "A::fun1" << '\t';
      fun2( );
    }
    void fun2( )
    { cout << "A::fun2" << '\t';
      fun3( );                                    //E
    }
    virtual void fun3( )                          //F  虚函数
    { cout << "A::fun3" << '\t';
      fun4( );
    }
    void fun4( )
    { cout << "A::fun4" << '\n'; }
};
class B: public A
{
public:
    void fun3( )
    { cout << "B::fun3" << '\t';
      fun4( );
    }
    void fun4( )
    { cout << "B::fun4" << '\n'; }
};
void main( )
{   A  a;
    a.fun1( );
    B  b;
    b.fun1( );                                    //G
}
```

程序的运行结果是：

```
A::fun1    A::fun2        A::fun3        A::fun4
A::fun1    A::fun2        B::fun3        B::fun4
```

我们注意到程序中fun1( )和fun3( )是虚函数，而fun2( )和fun4( )不是虚函数。程序第一行的输出很好理解，第二行的输出较难理解。类B继承了基类A的四个函数，b.fun1( )首先调用的是从基类继承下来的

fun1( )，fun1( )调用的仍然是从基类继承下来的fun2( )，那么fun2( )中调用的是从基类继承下来的fun3( )还是派生类中的fun3( )呢？我们注意，在成员函数中调用成员函数时，系统都是通过对象自身的指针this调用的，A类中的fun2( )的实际被处理成如下形式：

```
void fun2( )
{ cout << "A::fun2" << '\t';
  this->fun3( );                    //E
}
```

此时，this应是基类A类型的指针。在执行G行时，会将指向派生类对象b的指针传递给fun1( )函数中的this指针，顺序调用fun1( )和fun2( )，在执行到E行时，this仍然指向派生类对象b，调用的自然就是派生类B的fun3( )函数了。本例中this的指针类型是基类的，但其指向派生类对象，所以遵循虚函数的调用规则。

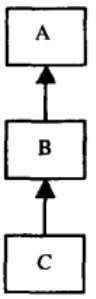


图13-4 例13.24中的继承关系

注意：在本例中，如果不把基类的fun1( )和fun3( )定义成虚函数，也就是说，把D行和F行的virtual关键字去掉，则程序的运行结果是：

```
A::fun1 A::fun2 A::fun3 A::fun4
A::fun1 A::fun2 A::fun3 A::fun4
```

**例13.24** 在构造函数中调用虚函数（见图13-4）。

```
#include <iostream.h>
class A
{
public:
    A( ) { fun( ); }
    virtual void fun( )                          //D   虚函数
    { cout << "A::fun" << '\t'; }
};
class B: public A
{
public:
    B( ) { fun( ); }
    void fun( )                                  //虚函数，缺省关键字virtual
    { cout << "B::fun" << '\t'; }
    void g( ) { fun( ); }                        //E
};
class C: public B
{
public:
    C( ) { fun( ); }
    void fun( )                                  //虚函数，缺省关键字virtual
    { cout << "C::fun" << '\n'; }
};
void main( )
{   C  c;                                        //F
    c.g( );                                      //G
}
```

程序的运行结果是：

```
A::fun    B::fun  C::fun
C::fun
```

第一行的输出是程序在F行中定义对象c时产生的。由于类的派生关系是A派生出B，B派生出C，定

义C类对象时，自然按顺序执行A、B、C三个类的构造函数，**虽然在B、C的构造函数中没有显式地调用其直接基类的构造函数，但编译器会自动调用基类的缺省构造函数**。在A、B、C三个类的构造函数中均调用了虚函数fun( )，C++规定它们调用的是各自类的虚函数，所以有第一行的输出。至于第二行为什么输出的是“C::fun”而不是“B::fun”，道理和例13.23中的一样，可将B类中的g( )函数理解为:

```
void g( ) {  this->fun( );  }
```

此时，this应是B类型的指针。G行的调用c.g( )，将派生类对象c的指针传递给了基类B的指针this，即基类指针指向派生类对象，由于fun( )是虚函数，因而调用的自然是C类的fun( )函数。如果将D行的virtual去掉，则程序的运行结果如下:

```
A::fun     B::fun     C::fun
B::fun
```

当派生类中函数的原型与基类中的虚函数原型一致时，在派生类中的才是虚函数，否则在派生类中的函数是重载函数，而不是虚函数，请看下例。

**例13.25**　比较虚函数和重载函数。

```
#include<iostream.h>
class A
{
public:
     virtual void fun()
     { cout<<"A::fun"<<'\n';}
};
class B: public A
{
public:
     void fun(int x=0)// 重载函数
     { cout<<"B:: fun"<<'\n';}
};
void main()
{    B b;
     A* pa=&b;
     pa->fun();
     b.fun();
}
```

运行结果: A::fun
　　B::fun

```
#include<iostream.h>
class A
{
public:
     virtual void fun()
     { cout<<"A::fun"<<'\n';}
};
class B: public A
{
public:
     void fun()  // 虚函数
     { cout<<"B:: fun"<<'\n';}
};
void main()
{    B b;
     A* pa=&b;
     pa->fun();
     b.fun();
}
```

运行结果: B::fun
　　B::fun

两个方框中的程序运行结果不同，请读者自行分析原因。

### 13.4.2　虚析构函数

前面已讲，如果类的构造函数中有动态申请的存储空间，则在析构函数中应释放该空间。此时，建议将析构函数定义为虚函数，以便实现通过基类的指针或引用释放派生类对象动态空间的多态性。

**例13.26**　虚析构函数（见图13-5）。

```
#include <iostream.h>
#include <string.h>
class base
{
   char *baseptr;
```

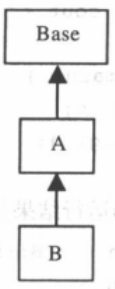



图13-5　例13.26中的继承关系

```
public:
  base( )
  {      baseptr=new char[100];
         strcpy(baseptr, "In class base");
         fc( );
  }
  virtual void fc( )                                    //D
  { cout<<baseptr<<endl; }
  virtual ~base( )                                      //E
  {     delete [ ]baseptr;
        cout<<"Delete [ ]baseptr"<<endl;
  }
};
class A : public base
{
   char *Aptr;
public:
   A( )
   {     Aptr=new char[100];
         strcpy(Aptr, "In class A");
         fc( );                                         //F
   }
   void f( ) { fc( ); }                                 //G
   ~A( )
   {     delete [ ]Aptr;
         cout<<"Delete [ ]Aptr"<<endl;
   }
};
class B : public A
{
   char *Bptr;
public:
  B( )
  {      Bptr=new char[100];
         strcpy(Bptr, "In class B");
  }
  void fc( ) { cout<<Bptr<< endl; }
  ~B( )
  {      delete [ ]Bptr;
         cout<<"Delete [ ]Bptr"<<endl;
  }
};
void main(void)
{  B b;                                                 //H
   b.f( );                                              //K
   base *p=new B;                                       //M
   delete p;                                            //N
}
```

程序的输出结果是:

```
In class base
In class base
In class B
In class base
```

```
In class base
Delete [ ]Bptr
Delete [ ]Aptr
Delete [ ]baseptr
Delete [ ]Bptr
Delete [ ]Aptr
Delete [ ]baseptr
```

在本例中，由基类base派生出A类，再由A类派生出B类。注意每个类的构造函数中都动态申请了存储空间，在析构函数中释放该空间。在D行将函数fc( )说明成虚函数，在E行将析构函数也说明成了虚函数。

在程序的H行，定义B类对象b，依次调用base类、A类、B类的构造函数。在base类的构造函数中，调用虚函数fc( )，输出"In class base"。在A类的构造函数中，调用虚函数fc( )，由于它自己没有fc( )函数，所以调用的是基类的fc( )函数，仍然输出"In class base"。注意在构造函数中调用虚函数，遵循静态联编规则。在B类的构造函数中，没有输出。

在程序的K行，调用的是从A类继承来的f( )函数，由于在程序的G行，f( )函数中调用的是虚函数fc( )，此时调用的是B类的fc( )函数。原因与例13.24中的一样。

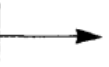
在程序的M行，定义了一个基类指针p，它指向一个动态申请的派生类对象，在生成该动态对象时，同理按顺序调用各基类的构造函数。

在程序的N行，释放该动态对象，由于将析构函数定义成了虚函数，所以按顺序调用B类、A类、base类的析构函数，释放每个类的动态申请的空间。

最后，在main( )函数结束前，撤销对象b，同样按顺序调用基类的析构函数。

值得注意的是程序的M行和N行，基类指针p指向派生类对象，对p所指向的派生类对象的处理，遵循处理派生类对象的一切规则。如果不把析构函数定义成虚函数，即删去程序E行中的关键字virtual，那么在释放p指向的对象时，因p是base类指针，则只调用基类base的析构函数。运行结果如下：

```
In class base
In class base
In class B
In class base
In class base
Delete [ ]Bptr
Delete [ ]Aptr
Delete [ ]baseptr
Delete [ ]Bptr
Delete [ ]Aptr
Delete [ ]baseptr
```




与前面的运行结果比较，少了两行，这就意味着，在释放M行动态申请的B类对象时，只释放了基类base的动态空间，B类和A类动态申请的空间没有被释放。这样，程序中就存在着潜在的错误。

如果把析构函数定义成虚函数，尽管p指向的是派生类对象，但是仍然遵循派生类对象的行为特点，即撤销派生类对象时，依次由低向高调用析构函数。这样就不会出现问题了。

将析构函数定义为虚函数的意义在于，通过基类指针（该基类指针指向派生类对象）或通过基类的引用（用派生类对象初始化该引用）释放派生类对象时，依次调用继承链上的各析构函数。如果直接定义派生类对象，则系统在撤销派生类对象时，会自动依次调用继承链上的各析构函数，而不需要将析构函数定义成虚析构函数。

## 13.5　纯虚函数和抽象类

在定义一个基类时，会遇到这样的情况：无法定义类中某些函数的具体实现。考虑例13.22有关“形

状”的类，从日常抽象思维的角度考虑，能否定义一个“形状”类Shape，对形状的通用操作可能有：求这个形状的面积、绘制这个形状的图形等。当然它是一个抽象的类，我们无法真正定义对这个形状操作的具体实现。可以由Shape类派生出具体的“点”类Point、“长方形”类Rectangle、“圆”类Circle等，在派生类中实现具体的求不同形状的面积和绘制不同形状的图形的操作。

我们可以把基类中无法有具体实现的函数定义为纯虚函数，一般格式为：

```
virtual <函数返回值类型> <函数名>( [<参数列表>] ) = 0 ;
```

我们注意到，该函数的定义没有函数体，即没有函数的具体实现部分。另外，函数参数列表圆括号后面的“= 0”，表示将函数名的值赋予0，就是将指向函数体的指针赋初值0，它无法调用任何函数。这与将虚函数定义成空函数是有区别的，如空函数的定义如下：

```
virtual <函数返回值类型> <函数名>( [<参数列表>] )
{ }
```

表示函数名是有指向的，调用该函数后，在函数体中不做任何事情就返回。

**纯虚函数**就是没有函数体（具体实现）的虚函数，而包含有纯虚函数的类称为**抽象类**。

改写例13.22为例13.27。

**例13.27** 定义抽象类，派生出若干类，在派生类中实现纯虚函数（见图13-6）。

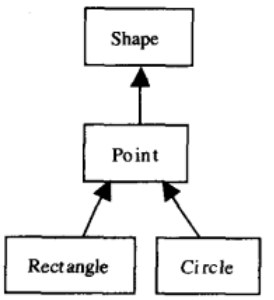



图13-6 例13.27中的继承关系

```
#include <iostream.h>
class Shape
{
public:
        virtual double Area( )=0;                    // 抽象的求面积的纯虚函数
        virtual void Draw( )=0;                      // 抽象的绘制图形的纯虚函数
};
class Point:public Shape
{
protected:
        double x, y;                                 // 点的坐标值
public:
        Point(double a=0, double b=0)
        { x=a; y=b; }
        double Area( )                               // 基类纯虚函数Area( )的具体实现
        { return 0.0; }
        void Draw( )                                 // 基类纯虚函数Draw( )的具体实现
        { cout <<"Draw Point!\n"; }
};
class Rectangle:public Point
{
protected:
        double x1, y1;                               // 长方形右下角点的坐标值，基类中x，y为左上
                                                        角坐标点
public:
        Rectangle(double a=0, double b=0, double c=0, double d=0):Point(a, b)
        { x1=c; y1=d; }
        double Area( )                               // 基类纯虚函数Area( )的具体实现
        { return (x-x1)*(y-y1); }
```

```
        void Draw( )                                    // 基类纯虚函数Draw( )的具体实现
        { cout <<"Draw Rectangle!\n"; }
};
class Circle:public Point
{
protected:
        double r;                                       // 半径，基类中x，y为圆心坐标点
public:
        Circle(double a=0, double b=0, double c=0):Point(a, b)
        { r=c; }
        double Area( )                                  // 基类纯虚函数Area( )的具体实现
        { return 3.14*r*r; }
        void Draw( )                                    // 基类纯虚函数Draw( )的具体实现
        { cout <<"Draw Circle!\n"; }
};
double CalcArea(Shape &s)
{       return(s.Area( ));  }                           // A   通过基类对象的引用实现动态多态
void DrawShape(Shape *sp)
{       sp->Draw( );    }                               // B   通过基类对象的指针实现动态多态
void main( )
{       Point p(1, 2);
        Rectangle r(0, 0, 1, 1);
        Circle c(0, 0, 1);
        cout<<CalcArea(p)<<'\t'<<CalcArea(r)<<'\t'<<CalcArea(c)<<'\n';
        DrawShape(&p);
        DrawShape(&r);
        DrawShape(&c);
}
```

程序的运行结果是:

```
0       1       3.14
Draw Point!
Draw Rectangle!
Draw Circle!
```

本例中Shape是抽象类，由它派生出Point类，再由Point类派生出Rectangle、Circle类。在三个派生类中，均有对基类中定义的两个纯虚函数的实现，此时派生类就不再是抽象类了。使用抽象类和纯虚函数的好处是，在基类中提供调用各派生类函数的统一接口，如Area( )和Draw( )函数，而各派生类中具有统一接口的函数Area( )和Draw( )有不同的具体实现方法。这样，利用C++提供的动态多态的特性，就可以实现编制通用函数的目的。如本例中CalcArea(Shape &s) 和DrawShape(Shape *sp)就是计算面积和绘制图形的通用函数，函数中的A行和B行，在函数调用时，通过基类的统一调用接口调用的是不同的函数。另外，注意通用函数CalcArea(Shape &s)，函数的形参s是Shape类对象，是实参对象（Shape的派生类对象）的引用。在此函数中计算s的面积。而在例13.22中的函数CalcArea(Point &p)，计算基类Point的对象p的面积，虽然也能通过动态多态实现求派生类Rectangle和Circle的对象的面积，但编写程序时仍然感觉不太自然，因为实参是Rectangle和Circle的对象，而形参是类Point对象，与日常思维不一致。使用抽象类编制的例13.24中的程序与日常思维逻辑保持一致。即计算面积的公用函数double CalcArea(Shape &s)，形参是Shape类对象，实参是不同的派生类形状的对象。

使用抽象类中的纯虚函数提供“统一的方法”，利用C++提供的动态多态特性，达到用统一的调用方法得到“不同的实现”的目的。

有关纯虚函数和抽象类的使用，注意以下两点：

（1）抽象类只能做派生类的基类。不能定义抽象类的对象。

（2）在抽象类的派生类中，如果没有对基类的纯虚函数的定义实现函数，那么它仍然是抽象类。

下面通过两个较复杂的例子，说明抽象类的精彩应用。

***例13.28** 定义一个双向链表，实现插入结点、删除结点、查找结点、输出链表中全体结点的操作。

分析：对任何链表的操作，如插入结点、删除结点、查找结点等的操作都是相同的，不同点在于各种类型的链表，其数据结点的值不同。如一个整型数据的链表，其结点是包含一个整型值的结构体变量；一个学生数据的链表，其结点是包含一个学生数据的结构体变量。

本例中链表结构如图13-7所示。

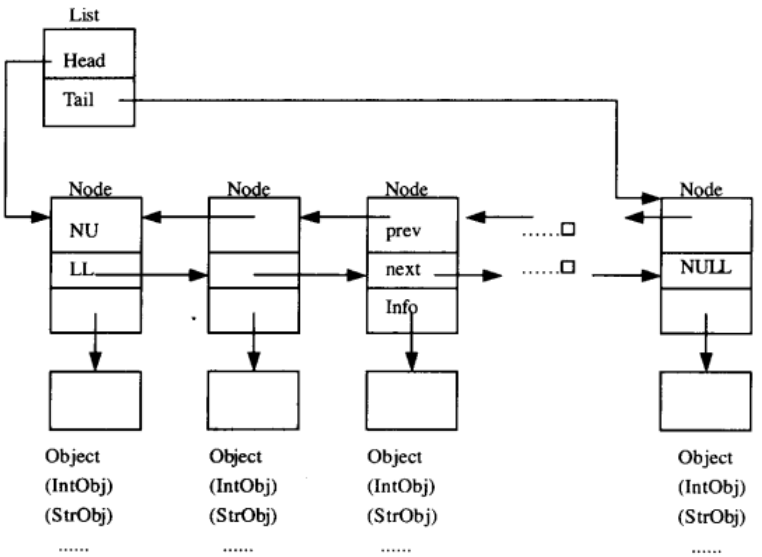



图13-7 使用抽象类产生的双向链表结构

通用的链表操作，涉及三个类。第一个为抽象类Object，它是数据结点的抽象描述，对它的操作只有两个，一是比较两个Object结点值是否相等，二是显示该结点的详细数据，这两个操作都定义为纯虚函数，其真正的实现在派生类中。如果希望实现一个具体的链表，就必须由抽象类Object派生出具体的数据结点，如本例和稍后要介绍的例13.29，由Object派生整型结点IntObj和字符串结点StrObj，以实现由整型结点构成的链表和由字符串结点构成的链表。当然，我们可以从Object类派生出任何我们想实现的链表的数据结点结构。第二个类是Node，它是构成链表的主结点，包含三个指针成员，分别指向前一结点、指向后一结点、指向数据结点Object。第三个类是List，它是链表类，包含指向链表首结点和指向尾结点的指针。List类是Node类的友元类，表示在List类的所有成员函数中均可以访问Node类的私有成员，以方便操作。

我们将上述三个类的定义放在头文件Li1328.h中，将派生类IntObj及主函数放在源程序文件Li1328.cpp中。

```
//Li1328.h                                    // 链表的通用操作部分
#include <iostream.h>
#include <string.h>
class Object                                  // 数据结点抽象类，用于派生实际数据结点
{
public:
       Object( ){}
       virtual int IsEqual(Object &)=0;       // 纯虚函数，实现两个数据结点比较
```

```
        virtual void Show( )=0;                    //纯虚函数，输出结点上的数据
        virtual ~Object( ){}                       //析构函数是虚函数
};
class Node                                         //链表主结点类
{       Object *Info;                              //指向数据结点的指针
        Node *Prev, *Next;                         //指向前、后结点的指针
public:
        Node( ){ Info=NULL; Prev=NULL; Next=NULL; }
        Node(Node &node)                           //拷贝构造函数
        {    Info=node.Info;
             Prev=node.Prev;
             Next=node.Next;
        }
        void FillInfo(Object *obj)                 //使Node结点的Info指针指向数据结点
        { Info = obj; }
        friend class List;                         //List类是Node类的友元类
};
class List                                         //双向链表类
{
        Node *Head, *Tail;                         //链表首、尾指针
public:
        List( ){ Head=Tail=NULL; };                //置为空链表
        ~List( ){ DeleteList( ); }
        void AddNode(Node*);                       //在链表尾加入一个结点
        Node * DeleteNode(Node*);                  //从链表中摘除一个结点，返回被摘除的结点指针
        Node * LookUp(Object&);                    //在链表中查找指定结点
        void ShowList( );                          //输出链表中全部结点的值
        void DeleteList( );                        //释放链表中全部结点空间
};
void List::AddNode(Node * node)                    //在链表尾加入一个结点
{
        if(Head==NULL)                             //条件成立，为空链表，此时插入第一个结点
        {    Head=Tail=node;                       //链首和链尾均指向该结点
             node->Next=node->Prev=NULL;           //将该结点的前、后指针置为空指针
        }
        else                                       //链表不空，将该结点加入链表尾
        {    Tail->Next = node;                    //原链表尾结点的后指针指向该结点
             node->Prev=Tail;                      //该结点的前指针指向原链表尾结点
             node->Next=NULL;                      //该结点的后指针为空指针
             Tail=node;                            //将链表的尾指针指向新加入的结点
        }
}
Node * List::DeleteNode(Node * node)
{                                                  //从链表中摘除一个结点，返回被摘除的结点指针
     if(node==Head)                                //两者相等，表示待摘除的结点是首结点
          if(node==Tail)                           //此条件若成立，表示链表中只有一个结点
                   Head=Tail=NULL;
          else
          {        Head = node->Next;              //摘除首结点
                   Head->Prev = NULL;
```

```
        }
    else
    {                                          // 待摘除的结点不是首结点
        node->Prev->Next = node->Next;         // 置其前一结点的后指针
        if(node == Tail)                       // 待摘除的结点是尾结点
            Tail = node->Prev;                 // 重置链表尾指针
        else
            node->Next->Prev = node->Prev;     // 置其后一结点的前指针
    }
    node->Prev = node->Next = NULL;            // 将摘除结点的前、后指针置为空
    return(node);                              // 返回除结点的首指针
}
Node * List::LookUp(Object & obj)              // A在链表中查找指定结点
{   Node *pn=Head;
    while(pn)
    {   if(pn->Info->IsEqual(obj))             // B调用派生类的IsEqual(obj)函数
            return pn;                         // 若找到要查找的结点，则返回链表中该结点的指针
        pn = pn->Next;
    }
    return NULL;                               // 若找不到，则返回空指针
}
void List::ShowList( )                         // 输出链表中全部结点的值
{   Node *p = Head;
    while(p)
    {   p->Info->Show( );                      // C调用派生类的Show( )函数
        p=p->Next;
    }
}
void List::DeleteList( )                       // 释放链表中全部结点空间
{   Node *p1, *p2;
    p1=Head;
    while(p1)
    {   delete p1->Info;                       // 先释放数据结点的动态空间
        p2=p1;
        p1 = p1->Next;
        delete p2;                             // 再释放Node结点的动态空间
    }
}
//Li1328.cpp                                   // 派生类及测试函数
#include "Li1328.h"
class IntObj : public Object                   // 从抽象类Object派生整型数据结点类IntObj
{
    int data;
public:
    IntObj(int x=0){ data=x; }
    void SetData(int x){ data=x; }
    int IsEqual(Object &obj)                   // D 重新定义结点比较函数
    {   IntObj &temp = (IntObj &)obj;          // E 引用类型转换，obj是Object类对象
        return(data==temp.data);               // 若相等则返回1，否则返回0。
    }
```

```
        void Show( )                          // F 重新定义显示结点数据函数
        { cout << "Data=" << data << '\t'; }
};
void main(void)
{       IntObj *p;
        Node *pn, *pt ;
        List list;                            // 定义一个链表
        for(int i=1; i<5; i++)                // 建立一个含有4个结点的双向链表
        {
              p = new IntObj(100+i);          //动态建立一个数据结点，即IntObj对象，p指向它
              pn = new Node;                  // 动态建立一个链表主结点，pn指向它
              pn->FillInfo(p);                // 将主结点的数据结点指针指向数据结点
              list.AddNode(pn);               // 向链表中加入该结点
        }
        list.ShowList( ); cout << endl;       // 显示链表全部结点值
        IntObj da;                            // 构造一个数据结点da
        da.SetData(102);                      // 给数据结点赋值
        pn = list.LookUp(da);                 // G 在链表中查找数据结点与da相等的结点，
                                              // 并返回找到的主结点的指针（Node*类型）
        if(pn) pt=list.DeleteNode(pn);        // 从链表中摘除pn指向的结点，实际上pn==pt
        list.ShowList( ); cout << endl;
        if(pn) list.AddNode(pt);              // 将pt指向的结点重新加入链表
        list.ShowList( ); cout << endl;
}
```

程序的运行结果是:

```
Data=101        Data=102        Data=103        Data=104
Data=101        Data=103        Data=104
Data=101        Data=103        Data=104        Data=102
```

例中的IntObj是从抽象类Object派生而来的，可以根据实际数据结构的需要来定义从基类继承来的虚函数IsEqual( )和Show( )，见程序中的D行和F行。

再看程序中的G行，调用list.LookUp(da)函数，实参是IntObj类对象，跟踪到A行，形参是Object&类对象，看B行，调用IsEqual(obj)函数，实参是Object&类对象，再跟踪到D行，形参也是Object&类对象，看E行，`IntObj &temp = (IntObj &)obj;`，这是一个将Object&类对象obj转化成IntObj&类对象temp的语句。从G行到E行，参数经过了IntObj→Object&→Object&→IntObj&的转化过程，在E行的temp实际上就是最初G行中的对象da的引用。也就是说在IsEqual(obj)函数中比较的是实际数据对象。

Object类派生了IntObj类，在F行重新定义了Show( )函数，此时C行调用的就是派生类的Show( )了，显示的是派生类对象的值了，这样就实现了不同类的对象，使用相同的接口实现了不同的操作。

下面再看一例，建立数据结点为字符串的双向链表，数据结点也是从抽象类Object派生出来的。

***例13.29**　利用例13.28 中链表的通用算法，实现处理字符串的双向链表。

```
//Li1329.cpp                              // 派生类及测试函数
#include "Li1328.h"
class StrObj:public Object                // 从抽象类Object派生字符串数据结点类StrObj
{
    char *Sptr;
public:
    StrObj( ){ Sptr=NULL; }
    StrObj(char *s)
```

```
    {    if(s)
         {      Sptr=new char[strlen(s)+1];
                strcpy(Sptr, s);
         }
         else
                Sptr=NULL;
    }
    ~StrObj( ){ if(Sptr) delete[ ]Sptr; }
    void SetStr(char *);
    int IsEqual(Object &obj);
    void Show( );
};
void StrObj::SetStr(char *s)
{    if(Sptr)delete[ ]Sptr;
     if(s)
     {      Sptr=new char[strlen(s)+1];
            strcpy(Sptr, s);
     }
     else
            Sptr=NULL;
}
int StrObj::IsEqual(Object &obj)        // 重新定义结点比较函数
{     StrObj &temp = (StrObj &)obj;    // 类型转换
      return(strcmp(Sptr, temp.Sptr)==0);
}
void StrObj::Show( )                    // 重新定义显示结点数据函数
{    cout<<"String="<<Sptr<<endl;  }
void main(void)
{    StrObj *p;
     Node *pn, *pt;
     List list;
     char s[100];
     for(int i=1; i<5; i++)
     {      cout<<"Input String: ";
            cin.getline(s, 100);       // 输入一行字符串
            p = new StrObj(s);
            pn = new Node;
            pn->FillInfo(p);
            list.AddNode(pn);
     }
     list.ShowList( );
     StrObj da;
     cout<<"Input a Search String:";
     cin.getline(s, 100);              // 输出待查找字符串
     da.SetStr(s);
     pn = list.LookUp(da);
     if(pn) pt=list.DeleteNode(pn);    // 摘除该字符串
     list.ShowList( );
     if(pn) list.AddNode(pt);          // 重新加入该字符串
     list.ShowList( );
}
```

程序的运行结果是（其中画线部分是用户的输入）：

```
Input String: C++<Enter>
Input String: study<Enter>
Input String: visual<Enter>
Input String: program<Enter>
String=C++
String=study
String=visual
String=program
Input a Search String: study<Enter>
String=C++
String=visual
String=program
String=C++
String=visual
String=program
String=study
```

从此例再次看出，头文件Li1328.h中是链表操作的通用类及算法，不需要对它做任何改动，而只需要从抽象数据结点类派生出一个新的实际的数据结点类，即可实现建立一种新的数据结点的双向链表。

结论：无论抽象类Object的派生类的数据结构如何变化，类Node和List均不要做任何修改，充分体现了OOP技术所支持的代码重用性。使用虚函数可以实现通用程序设计。VC++的基础类库MFC的建立就是使用了这种技术，在基础类库中建立了大量的抽象类，供用户派生出适合自己应用的类，重用系统提供的代码；用户站在“巨人”的肩膀上，很容易实现自己的应用。

## 习题13

### 一、简答题

1. 什么是运算符重载？实现运算符重载有几种方法？如何实现？
2. 如何实现本类对象与其他类对象之间的相互类型转换？
3. 什么是虚函数？什么是纯虚函数？它们的作用是什么？
4. 试比较虚函数与虚基类的概念及作用。

### 二、选择题

1. 编译器无法根据下列条件中的_____区分调用哪一个重载函数。

   A）函数参数的个数　　B）函数参数的类型

   C）函数的返回值类型　　D）是否有参数

2. 下列函数中_____不能重载。

   A）成员函数　　B）非成员函数　　C）构造函数　　D）析构函数

3. 下列关于重载函数的描述中，_____是错误的。

   A）重载函数中不允许使用缺省参数

   B）编译器根据重载函数的参数表选择执行不同的重载函数

   C）不要使用重载函数来描述毫不相干的函数

   D）构造函数的重载将会给初始化带来多种方式

4. 下列关于运算符重载的描述中，_____是正确的。

   A）运算符重载可以改变操作数的个数　　B）运算符重载可以改变优先级

   C）运算符重载可以改变结合性　　D）运算符重载不可以改变语法结构

5. 关于动态联编的描述中，_____是错误的。

A）动态联编是以虚函数为基础的

B）动态联编是在编译时确定所调用的函数的

C）动态联编是通过对象的指针或对象的引用调用虚函数来实现的

D）动态联编是在运行时确定所调用的函数的

6. 下列关于虚函数的描述中，_____是正确的。

A）虚函数是一个非成员函数

B）虚函数是一个static类型的成员函数

C）派生类的虚函数与基类的虚函数具有不同的参数个数和类型

D）基类中说明了虚函数后，派生类中与其对应的虚函数可以不加virtual说明

7. 以下有关析构函数的描述中，_____不正确。

A）析构函数可以被定义成纯虚函数　　B）一个类中只能定义一个析构函数

C）析构函数不能有参数　　D）析构函数的定义只能在类体内

8. 关于纯虚函数和抽象类的描述中，_____是错误的。

A）纯虚函数是一种特殊的虚函数，它没有具体的实现。

B）具有纯虚函数的类是抽象类。

C）一个基类中说明有纯虚函数，该基类的派生类一定不再是抽象类。

D）抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。

**三、运行结果、完善程序**

1. 请写出下述程序的运行结果:

```
#include <iostream.h>
class Sample
{
    int n;
public:
    Sample(int i=0){ n=i; }
    Sample operator++(){ n++; return *this; }
    Sample operator++(int){ Sample t=*this; n+=2; return t; }
    operator int( ) { return n; }
    void show( ){ cout<<"n="<<n<<endl; }
};
void main( )
{   Sample A(2),B(2);
    int a,b;
    a=A++;
    b=++B;
    A.show( );
    B.show( );
    cout<<a<<','<<b<<endl;
}
```

2. 请写出下述程序的运行结果:

```
#include <iostream.h>
class B
{
private:
     int b;
public:
     B(int i=0){ b=i; }
```

```
    virtual void show( )        // A行
    {    cout<<"B::show( ) called. "<<b<<endl; }
};
class D: public B
{
private:
    int d;
    void show( )
    {    cout<<"D::show( ) called. "<<d<<endl; }
public:
    D(int i=0, int j=0):B(i)
    { d=j; }
};
```

```
void fun(B*obj)
{   obj->show(); }
void main()
{
    D*pd=new D(5,8);
    fun(pd);
    delete pd;
}
```

```
void fun(B & obj)
{  obj.show(); }
void main()
{
    D*pd=new D(5,8);
    fun(*pd);
    delete pd;
}
```

先看虚线框中的程序，运行结果为: _____

若将A行的virtual去掉，则结果为: _____

对本题，若将虚线框中的fun( )和main( )函数同时修改成右侧实线框中的对应函数，则若A行有virtual时，结果为: _____；若A行无virtual时，结果为: _____。

3. 请写出下述程序的运行结果:

```
#include <iostream.h>
class B
{
protected:
    char ver;
public:
    B( ){ ver='B'; }
    virtual void print( )                    // A行
    {    cout<<"The B version "<<ver<<endl;   }
};
class D1: public B
{
private:
    int info;
public:
    D1(int number){ info=number; ver='1'; }
    void print( )
    {    cout<<"The D1 info: "<<info<<" version "<<ver<<endl;      }
};
class D2: public B
{
private:
```

```
        int info;
public:
        D2(int number){ info=number; }
        void print( )
        {       cout<<"The D2 info: "<<info<<" version "<<ver<<endl;    }
};
class D3: public D1
{
private:
        int info;
public:
        D3(int number):D1(number)
        {    info=number;
             ver='3';
        }
        void print( )
        {     cout<<"The D3 info: "<<info<<" version "<<ver<<endl;     }
};
void print_info(B *p)
{       p->print( );  }
void main( )
{       B b;
        D1 d1(4);
        D2 d2(100);
        D3 d3(-25);
        print_info(&b);
        print_info(&d1);
        print_info(&d2);
        print_info(&d3);
}
```

运行结果为_____。

若将A行的virtual去掉，则运行结果为_____。

4. 下面程序实现了将类Num的对象转换成一个字符串（本质上是转换成一个字符指针，指向字符串的首字符）。如果类Num的对象成员数据为整数2468，则转换得到的字符串为“2468”。请完善程序。

```
#include <iostream.h>
class Num
{
   int n;
public:
   Num(int i=0){ n=i; }
   void show( ){ cout<<"n="<<n<<endl; }
      _____(1)_____;      // 类型转换函数原型说明
};
Num::operator char *( )
{ int i=0,t=n;
  while(t>0) i++,t=t/10;
  char *s=new char[i+1];
  s[i]='\0';
  t=n;
```

```
    for(i--; i>=0; i--)
    {       s[i]='0'+t%10;
            t=t/10;
     }
            (2)                    ;
}
void main( )
{
   Num num=2468;
   char *str;
            (3)                    ;        // 将对象num转换成字符串，令str指向该字符串
   num.show( );
   cout<<"str="<<str<<endl;
   delete [ ]str;
}
```

**四、编程题**

1. 定义一个复数类，其数据成员为复数的实部和虚部，用成员函数重载运算符=、+=，用友元函数重载运算符+、－、直接实现两个复数间的各种运算。编写一个完整的程序，测试各种重载的正确性。
2. 定义一个分数类，其数据成员为分子和分母。用成员函数重载运算符 =、+和－，以实现分数的赋值、相加和相减；用友元函数重载运算符*和/，以实现分数的相乘和相除。要求进行各种运算后，得到的分数是最简的。提示：为了得到最简分数可编写一个成员函数用于约简分子和分母，进而要编写一个求两个数的最大公约数的函数。请编写一个完整的程序，测试各种重载的正确性。假定分子和分母都是正整数。
3. 定义描述平面上一个点的类Point，重载“++”和“--”运算符，要求用成员函数实现“++”的前置和后置运算；用友元函数实现“--”的前置和后置运算。编写一个完整的程序，测试各种运算。
4. 建立一个字符串类String，要求如下：

   **私有数据成员**：

   - char *str:　字符串的首地址
   - int len:　该字符串的长度值

   **公有成员函数**：

   - 构造函数：动态申请存储字符串所需的内存空间，并既能用指定的字符串也能用默认的值0为所声明的对象进行初始化。
   - 拷贝构造函数。
   - 析构函数：释放对象存储字符串所用的内存空间。
   - void set(char *s)：为该对象赋值。
   - void show( )：显示该对象中保存的字符串。
   - int getlen( )：返回该对象的字符串长度值。
   - void delchar(char ch)：在该对象保存的字符串中查找并删除指定的所有字符。
   - String & operator =(String &)：重载“=”运算符，实现字符串的直接赋值。
   - String & operator +=(String &)：重载“+=”运算符，实现两字符串的拼接。
   - friend String operator +(String &, String &)：重载“++”运算符，实现两字符串的拼接。
   - int operator ==(String &)：重载“==”运算符，进行两字符串的相等性比较，相等则返回1，否则返回0。

   编写完整的程序，测试类的各成员函数的正确性。
5. 定义一个矩阵类Matrix，其数据成员为：int *p, m, n;，用于存放二维数组值。要求将二维数组存放成一维数组，m和n表示二维数组的行数和列数，p指向数组第一个元素。要求编写以下几个公有成员函数。

- Matrix(int mm, int nn )：构造函数，初始化数组的行数和列数，并动态为数组申请空间。
- Matrix (Matrix &b)：拷贝构造函数。
- void input( )：输入数组值。
- Matrix operator +(Matrix &b)：重载加法运算符“+”，实现两个矩阵相加。
- void operator =( Matrix &b)：重载赋值运算符“=”，实现两个矩阵赋值。
- void show( )：按二维方式输出a数组。
- ~Matrix( )：析构函数，释放数组空间。

在主函数中完成对该类的测试。首先输入m和n的值，构造三个矩阵类对象A、 B和C，输入矩阵A和B的值，完成运算C=A+B。自行设计测试用例。

6. 定义一个包含两个纯虚函数的抽象类Shape:

```
class Shape
{public:
    virtual float Area(void)=0;
    virtual void SetData(float, float=0)=0;
};
```

成员函数SetData( )用于设置几何图形的属性值。成员函数Area( ) 用于求出几何图形的面积。由Shape类派生出四种几何图形，它们分别是：三角形、矩形、正方形和圆形。例如三角形类的派生类如下:

```
class Triangle:public Shape
{    float w, h;
public:
     Triangle(float ww=0, float hh=0)
{    w=ww; h=hh; }
     float Area(void) {  return w*h/2; }
     void SetData(float ww, float hh=0) { w=ww; h=hh; }
};
```

其他图形的派生类请自行编写。派生类的类名和属性如下:

(1) 类名：Triangle　　属性：三角形一边长w和此边上的高h

(2) 类名：Rectangle　　属性：矩形的长w和宽h

(3) 类名：Square　　属性：正方形边长s

(4) 类名：Circle　　属性：圆的半径r

要求利用虚函数实现多态，编写一个求几何图形面积CalcArea( )的通用函数，其参数为抽象类Shape的指针。

在主函数中定义四种几何图形的对象，调用CalcArea( )求四种几何图形对象的面积之和。

# 第14章 输入／输出流

## 14.1 输入／输出基本概念

### 14.1.1 输入／输出流

计算机程序在执行时被调入内存，此时将程序称为内部程序。程序在执行过程中需要从外部设备输入一些数据，处理后将结果输出到外部设备中。对输入而言，外部设备可能是键盘和数据文件等；对输出而言，外部设备可能是显示器、打印机和数据文件等。这些外部设备有一些对应于真正的物理设备，有一些不是物理设备（如数据文件）。所以有时我们将外部设备称为逻辑设备。

数据在外部设备和内存之间传递就像水流一样，所以我们形象地称之为数据流，如图14-1所示。

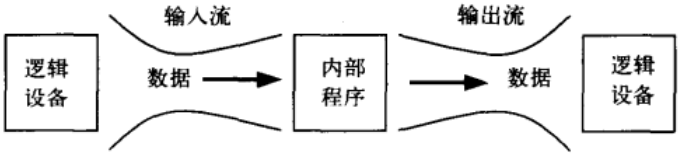



图14-1 输入／输出流

### 14.1.2 文本流、二进制流和数据文件

数据流分成文本流和二进制流，分别对应于文本文件和二进制文件。在文本流中，我们可以把数据看成一串ASCII字符，具体表现为一串ASCII码。对于数据12345，如果把它放进文本流中，它就被看成字符串“12345”，在流中的信息为一串ASCII码，其十六进制形式和二进制形式如图14-2所示。

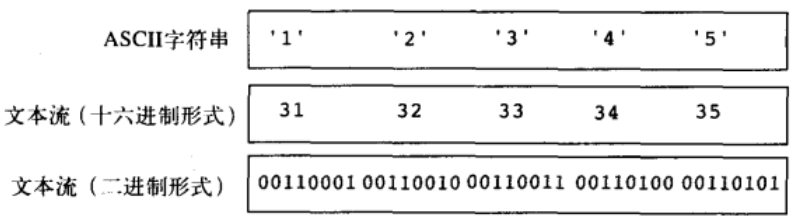



图14-2 文本流示意

在文本文件中，文本数据12345占用5个字节。

例如用Windows操作系统的记事本编辑器创建的文件就是以文本流的方式存放在文本文件中的。再如我们学习的VC编辑器，用它建立的源程序文件（.cpp文件）也是文本文件。

在二进制流中的数据是其内存映像，也就是说数据在内存中的表示形式是什么，它在数据流中的形式就是什么。例如数据12345，在VC++ 6.0环境中，如果我们把它表示成一个整数，则其在内存中的形式和二进制流中的形式相同，如图14-3所示。

在二进制文件中，二进制数据12345占用4个字节。

在数据流中，我们以字节为单位来衡量数据量的多少。

### 14.1.3 缓冲

在进行输入／输出时，计算机中央处理器CPU要跟外部设备打交道。每输入／输出一次，CPU都要

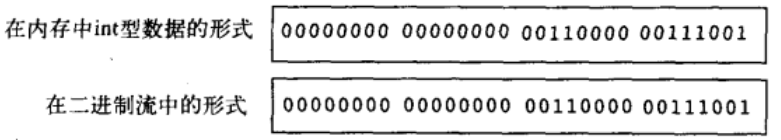



图14-3 二进制流示意

驱动一次外部设备，但是外部设备的操作速度与CPU的运行速度相比是非常慢的。因此，计算机的输入／输出处理系统，采用了“缓冲”技术来解决这个问题。以输出操作为例，在内存中开辟一个输出缓冲区，程序中的每次输出，都不是立刻直接输出到外部设备上的，而是首先输出到缓冲区中，当缓冲区满后或缓冲区不满但程序干涉时，才将缓冲区中的数据成批送往输出设备，这样可以减少对外部设备的驱动次数，提高输出速度。

引入缓冲的目的是解决CPU的运行速度和外部设备操作速度不匹配的矛盾。先将数据写入缓冲区，再成批输出。成批写出数据比一个一个写出数据节约时间。带缓冲的输入／输出系统如图14-4所示。

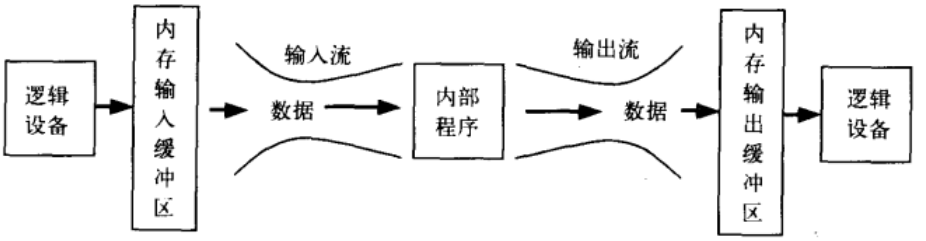



图14-4 带缓冲的输入／输出流

从图14-4中我们可以看出，在缓冲输入／输出系统中，CPU不直接跟外部设备交换数据，而是通过内存缓冲区与外部设备交换数据。CPU的速度与内存的存取速度相差不大，所以基本上进行输入／输出操作，不影响程序的执行速度。当然，有时需要非缓冲输入／输出系统。如在输出出错信息时，使用非缓冲系统更好。

## 14.2 输入／输出流类体系

### 14.2.1 基本输入／输出流类体系

在C++中，系统预先定义了很多类，给用户提供一些基本的通用的基础程序，用户只要搞清楚这些类的功能，直接使用即可，这样可以节省很多软件开发的时间。对于输入／输出操作，系统也预定义了有关的“类”，我们称之为输入／输出流类，用于完成数据的“流动(输入／输出)”。C++的基本流类体系如图14-5所示。

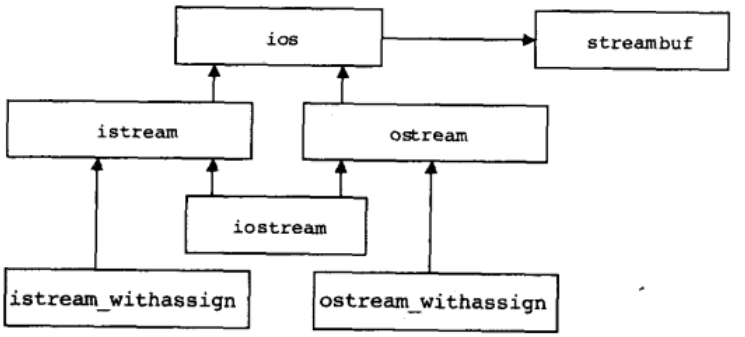



图14-5 基本输入／输出流类体系

图14-5中ios是所有输入／输出流类的基类，它是在头文件ios.h中被定义的，其他的类均是从该类派生出来的。通常我们在源程序的开头总是要包含头文件“iostream.h”，而在文件iostream.h中又包含了其他的头文件，其中包括ios.h、istream.h和ostream.h。注意图14-5中streambuf不是ios类的派生类。ios类中有一个指针成员，它指向一个streambuf类的对象。streambuf类的作用是管理输入／输出流类的缓冲区。

### 14.2.2　用运算符重载实现标准设备的输入／输出

我们将键盘称为标准输入设备，将显示器称为标准输出设备。在第3章介绍了有关标准输入／输出设备的简单输入／输出操作，如intx,y;cin>>x>>y;。这种操作与本章介绍的基本输入／输出流类体系有什么关系呢？我们观察一下头文件istream.h中的部分内容：

```
class istream : virtual public ios {
public:
      ......
      istream& operator>>(char *);                    //A 字符串输入重载函数
      ......
      istream& operator>>(char &);                    //B 字符输入重载函数
      ......
      istream& operator>>(int &);                     //C 整数输入重载函数
      ......
      istream& operator>>(double &);                  //D 实型数据输入重载函数
      ......
};
......
class istream_withassign : public istream {
......
};
extern istream_withassign cin;
......
```

可以看出，istream类公有继承ios类，而且ios类是虚基类。istream_withassign类公有继承istream类，cin是istream_withassign类的一个对象。右移运算符>>原本是位运算符，它的运算结合性是自左至右的。在istream类中将运算符>>重载为提取运算符。在istream类中对C++的基本数据类型均定义了运算符重载函数，如程序中的A、B、C、D行。C++编译器会将cin>>x>>y; 解释成(cin.operator>>(x)).operator>>(y);，在重载函数内部由系统去实现输入操作，用户不必关心输入实现的细节。C++的所有的输入／输出函数都是基于这个原理。

**例14.1**　在istream类中，重载提取运算符>>的含义。

```
#include <iostream.h>
void  main( )
{     int x, y;
      (cin.operator>>(x)).operator>>(y); // 等价于: cin>>x>>y;
      cout <<x << '\t'<< y << endl;
}
```

我们注意到上述重载函数的返回值均为istream类的对象的引用，如A、B、C和D这四行。这是为了实现连续的输入操作，如上述输入语句，首先将cin>>x解释为cin.operator>>(x)，函数调用的返回值为cin自身，这样才可以继续下一步输入y的操作，等价于输入x后，再进行cin>>y。

除了上述输入流对象cin，C++还定义了三个输出流对象cout、cerr和clog。这三个对象均是ostream_withassign类的对象，它们均在头文件ostream.h中被定义，用于完成各种输出操作。对于运算符<<，它原本是位运算符中的左移运算，在ostream类的定义中，将其重载为完成输出的插入运算符。

cin是输入流对象，用来处理标准输入，即键盘输入。cout是输出流对象，用来处理标准输出，缺省

时，输出到屏幕上。cout是带缓冲的输出流。cerr是输出流对象，一般用来将出错信息输出到标准输出设备上。cerr处理不带缓冲区的输出，即遇到cerr输出的信息，立刻直接输出到屏幕上，而不是先送往缓冲区，然后再输出。clog也是输出流对象，输出到标准输出设备上，一般用于输出一些提示信息，或系统的日志信息，clog提供的是带缓冲区的输出。

**例14.2** 使用cout、cerr和clog。

```
#include <iostream.h>                                    // 此程序的源程序文件名是 Li1402.cpp
#include <math.h>
void  main( )
{   double x;
    cout<<"Please input x:";                             // A
    cin>>x;
    if(x<0)
    {    cerr<<x<<" is not a positive number!\n";        // B
         return;
    }
    clog<<"The root of "<<x<<" is "<<sqrt(x)<<endl;      // C
}
```

在例14.2中，cout、cerr和clog的作用相同。而cerr和clog的区别在于，cerr是非缓冲输出，而clog是缓冲输出。请看下述例子。

**例14.3** 体会cerr和clog的区别。

```
#include <iostream.h>
void  main( )
{    double x;
     clog<<"Please input x:";          // A
     cin>>x;
     cout<<x<<endl;                    // B
}
```

此程序在屏幕上的运行过程如下：

```
5<Enter>                    // 输入
5                           // B行的输出
Please input x:             // A行的输出
```

为什么A行的输出反而在B行的后面呢？原因是：clog是缓冲输出，执行A行时，将输出信息输出到缓冲区中，而没有在屏幕上显示；然后输入x，输出x，当程序执行结束时，系统会将输出缓冲区中的内容全部显示到屏幕上，而不管缓冲区是否已满。对此程序，可将A行的clog改为cout或改为cerr，程序就能正常运行了。

### 14.2.3 缺省的输入／输出格式

#### 1. 缺省的（默认的）输入格式

在使用cin和提取运算符>>输入数据时，应注意以下几个问题：

（1）cin是缓冲流，当输入一行结束按回车（Enter）时，操作系统将输入内容放入输入缓冲区，然后输入操作函数才开始从缓冲区提取数据。

（2）输入数据的类型必须与提取数据的变量类型一致，否则输入数据时会出错。但这种错误不中止程序的执行，而是将系统的出错状态标志字中的对应错误位置为1。例如：有语句`int x,y; cin>>x>>y;`，若在程序执行时输入：`e 15<Enter>`，则输入x时就出错了，此时系统处理成将x的值置为0，而y的值为随机数。程序继续执行。

（3）输入数据时，在缺省的情况下，数据之间的分隔符为：<Space>键（空格键）、<Tab>键（制表键）或<Enter>键（回车键），这三个键通称为空白字符。

如：`char c1, c2, c3; cin>>c1>>c2>>c3;`，程序运行时，可输入：a<Space>b<Tab>c<Enter>，也可输入：ab <Tab> c <Enter>，或输入：a <Enter> b <Enter> c <Enter>，在上述情况下，三个变量c1、c2和c3获得值均为 'a'、'b' 和 'c'。应注意，最后输入的总是回车键。

又如：`char s1[20], s2[20]; cin>>s1>>s2;`，在输入数据时，输入ab<Space>cd<Enter>或输入ab<Tab>cd<Enter>或输入ab<Enter>cd<Enter>，结果字符串s1和s2提取的值均分别为"ab"和"cd"。

我们可以看到，回车<Enter>起两个作用：（1）表示一行结束，可进行提取操作了。（2）可用它做输入数据的分隔符。

使用缺省方式输入数据有一定的局限，如输入字符型数据时，无法将空格作为一个字符输入；在输入字符串时，无法输入带空格的字符串。如`char s[20]; cin>>s;`，若运行时输入：

```
Good morning! <Enter>
```

则s1获得的值是"Good"而不是"Good morning!"。

稍后，我们会介绍如何改变输入格式。

### 2. 缺省的（默认的）输出格式

不同类型数据的缺省的输出格式不同：

（1）输出整型数：数制是十进制，域宽为0，右对齐，以空格填充。

（2）输出实型数：精度6位（即有效数字是6位数），浮点输出，域宽为0，右对齐，以空格填充。若整数部分超过7 位或有效数字在小数点后第4 位之后，则自动转为科学计数法格式输出。

（3）输出字符或字符串：域宽为0，右对齐，以空格填充。

注意，上述描述中的“域宽”是指当输出一个数时，该数应该占用的字符宽度。如果数的实际宽度超过了规定的“域宽”，则按实际宽度输出。所以“域宽为0”表示按实际长度输出。在以后的学习中，我们可用setw( )函数设置一个数据的输出域宽。如果数的实际宽度小于规定的“域宽”，则以设定的填充字符填充到与“域宽”相同的宽度。

**例14.4** 标准输出的默认输出格式。

```
#include <iostream.h>
#include <iomanip.h>
void  main( )
{    double d1=12.3456789, d2=123456.789, d3=0.0000123456;
     cout<<d1<<','<<d2<<','<<d3<<endl;
     cout<<setw(10)<<d1<<','<<setw(10)<<d2<<','<<setw(10)<<d3<<endl;
     char s[10]="abcd", c='k';
     cout<<s<<','<<setw(4)<<c<<endl;
}
```

例14.4的输出结果是：

```
12.3457,123457,1.23456e-005
□□□□12.3457, □□□□123457,1.23456e-005
abcd,□□□k
```

注意：□表示空格，在屏幕上显示为空白。setw(10)表示设定下一个输出数据的域宽为10。变量d1和d2的值的有效数字超过了6位，只输出了前6位。变量d1的实际输出域宽是7位（含小数点），而设定的输出域宽为10，因默认的对齐方式为右对齐，所以前面补三个空格。变量d3的实际输出域宽为12位，超过了设定域宽10，按实际域宽输出。

一般来说，使用缺省的输入／输出格式就可以了。但有时会有一些特殊要求，如希望输入／输出八进制数、十六进制数，希望按左对齐方式输出数据等，因此在稍后，我们会介绍如何改变输出格式。

## 14.3 输入／输出格式控制

使用缺省的输入／输出格式不能满足一切需要。我们可以通过系统提供的一些方法来设置自己所需要的输入／输出格式。

### 14.3.1 使用成员函数进行格式控制

cin和cout分别是输入和输出流对象，它们所属的类是ios类的派生类。在ios类中提供了若干方法用于控制输入／输出格式，ios类的公有派生类对象cin和cout当然也可以使用这些方法。

首先来看一看在ios.h中有关ios类的定义的部分内容:

```
class _CRTIMP ios {
public:
        ......
     enum {  skipws          = 0x0001,       // 二进制形式 0000 0000 0000 0001
             left            = 0x0002,       // 二进制形式 0000 0000 0000 0010
             right           = 0x0004,       // 二进制形式 0000 0000 0000 0100
             internal        = 0x0008,       // 二进制形式 0000 0000 0000 1000
             dec             = 0x0010,       // 二进制形式 0000 0000 0001 0000
             oct             = 0x0020,       // 二进制形式 0000 0000 0010 0000
             hex             = 0x0040,       // 二进制形式 0000 0000 0100 0000
             showbase        = 0x0080,       // 二进制形式 0000 0000 1000 0000
             showpoint       = 0x0100,       // 二进制形式 0000 0001 0000 0000
             uppercase       = 0x0200,       // 二进制形式 0000 0010 0000 0000
             showpos         = 0x0400,       // 二进制形式 0000 0100 0000 0000
             scientific      = 0x0800,       // 二进制形式 0000 1000 0000 0000
             fixed           = 0x1000,       // 二进制形式 0001 0000 0000 0000
             unitbuf         = 0x2000,       // 二进制形式 0010 0000 0000 0000
             stdio           = 0x4000        // 二进制形式 0100 0000 0000 0000
     };
        ......
     static const long basefield;            // dec | oct | hex
     static const long adjustfield;          // left | right | internal
     static const long floatfield;           // scientific | fixed
        ......
     inline long flags( ) const;
     inline long flags(long _l);
     inline long setf(long _f, long _m);
     inline long setf(long _l);
     inline long unsetf(long _l);
     inline int width( ) const;
     inline int width(int _i);
     inline char fill( ) const;
     inline char fill(char _c);
     inline int precision(int _i);
     inline int precision( ) const;
protected:
     int     state;                          // 输入／输出流出错状态字（其使用见14.8节）
     long    x_flags;                        // 输入／输出格式状态字
     int     x_precision;                    // 浮点数的精度
     char    x_fill;                         // 填充字符
     int     x_width;                        // 输出数据的域宽
        ......
};
```

首先注意类的保护的数据成员：x_flags、x_precision、x_fill和x_width，它们被派生类继承，因此对象cin和cout均拥有自己的一套该数据成员。在输入／输出时，系统根据这几个数据成员的值，来控制输入／输出格式。x_width 的值表示输出数据的域宽，即当输出一个数时，它所占用的最小的字符位数。x_fill的值是填充字符，即当一个数据实际的输出宽度小于域宽时，用该字符填充不足部分，使该数据的输出宽度达到域宽。x_precision的值是浮点数的精度。最复杂的是x_flags，它是一个长整型数据，共有32个二进制位，每一位都代表了一种输入／输出状态，值为1或为0表示不同的输入／输出状态。我们称x_flags为输入／输出状态字。比如它的最低位若设置成1，则表示输入时跳过（不提取）空白字符，空白字符就是我们前面讲过的输入数据之间的分隔符；若将最低位设置成0，则表示输入时不跳过空白字符，即提取空白字符，它的意义我们在后面将看到。

C++系统依据对数据成员x_flags、x_precision、x_fill和x_width的不同设置来控制不同的输入／输出格式。系统提供了若干公有成员函数，编程者可通过对象cin和cout调用这些公有成员函数，来对上述四个数据成员的值进行设置。

如对x_flags进行设置时，就要搞清楚它的每一个二进制位的含义，在指定某个二进制位时，如果直接用数值来指定哪一位，记忆起来比较困难。于是在ios类中定义了一个无名的公有的枚举类型数据成员，enum { skipws= 0x0001, .....}，每个枚举常量代表一个状态位，见上面的定义，引用这些枚举常量的方法是：ios::skipws、ios::left等。枚举常量的取名是有意义的，这是为了方便记忆。每一个枚举常量代表一个标志位，其意义见表14-1。从该表中可以看出，虽然x_flags是32位的，但实际只使用了它的低15位。

x_flags的某些位是不能同时设置的，如控制输入／输出数据的数制的标志位ios::dec、ios::oct和ios::hex。为了使用方便，在ios类中，还定义了下面的静态成员：

```
static const long basefield;   其值为 dec | oct | hex
static const long adjustfield; 其值为 left | right | internal
static const long floatfield;  其值为 scientific | fixed
```

引用这几个静态成员的形式是，ios::basefield、 ios::adjustfield和 ios::floatfield。

**表14-1　ios标志位的含义（表中O代表Output，I代表Input）**

| 标志位<br>(枚举常量) | 值 | 含　义 | 适用输入／输出 |
|---|---|---|---|
| skipws | 0x0001 | 跳过输入中的空白字符 | I(输入) |
| left | 0x0002 | 输出数据左对齐 | O(输出) |
| right | 0x0004 | 输出数据右对齐 | O |
| internal | 0x0008 | 输出数据的符号或表示数制的前导字符左对齐，数据本身右对齐，中间为填充字符 | O |
| dec | 0x0010 | 输入／输出十进制数 | I/O |
| oct | 0x0020 | 输入／输出八进制数 | I/O |
| hex | 0x0040 | 输入／输出十六进制数 | I/O |
| showbase | 0x0080 | 输出数据时带有表示数制的前导字符（0或0x） | O |
| showpoint | 0x0100 | 浮点数输出时带有小数点 | O |
| uppercase | 0x0200 | 用大写字母输出十六进制数 | O |
| showpos | 0x0400 | 输出正数前面的“+”号 | O |
| scientific | 0x0800 | 用科学表示法输出浮点数 | O |
| fixed | 0x1000 | 用定点表示法输出浮点数 | O |
| unitbuf | 0x2000 | 每次插入后，立即刷新输出流 | O |
| stdio | 0x4000 | 每次插入后，立即刷新stdout和stderr | O |

“|”是按位“或”运算，根据位运算的含义，ios::basefield的值是0x0070，它是三个常数按位“或”

运算的结果:

```
ios::dec            0x0010        00000000 00010000
ios::oct            0x0020        00000000 00100000
ios::hex            0x0040        00000000 01000000
ios::basefield      0x0070        00000000 01110000     //“或”
```

ios:: adjustfield和ios::floatfield的值可以类推。

对状态字x_flags进行设置的公有成员函数的意义见表14-2。

**表14-2 格式控制成员函数的意义**

| 成员函数原型 | 功 能 |
|---|---|
| long flags( ) | 返回当前标志字的值 |
| long flags(long) | 将参数设置为新的标志字，返回设置前的标志字 |
| long setf(long , long) | 将标志字的某些位置新值，第二个参数指定待设置的若干位，第一个参数指定这些位的新值，返回设置前的标志字 |
| long setf(long) | 将参数指定的标志位置为1，返回设置前的标志字 |
| long unsetf(long) | 将参数指定的标志位置为0，返回设置前的标志字 |
| int width( ) | 返回当前域宽（缺省域宽为0） |
| int width(int) | 用参数值设置新的域宽，返回设置前的域宽 |
| char fill( ) | 返回当前的填充字符（缺省填充字符为空格） |
| char fill(char) | 用参数值设置新的填充字符，返回设置前的填充字符 |
| int precision( ) | 返回当前的精度（缺省值为6） |
| int precision(int) | 用参数值设置新的精度，返回设置前的精度 |

下面我们举几个例子，说明怎样使用格式控制成员函数实现输入／输出格式控制。

**例14.5** 使用成员函数，控制输出格式。

```
#include <iostream.h>
void  main( )
{
   int x=165, y=142;                          // (165)10=(245)8=(a5)16,  (142)10=(216)8=(8e)16
   cout.setf(ios::oct, ios::basefield);       // A 设置输出数据为八进制数
   cout<<x<<','<<y<<endl;
   cout.setf(ios::showbase);                  // B
   cout<<x<<','<<y<<endl;
   cout.width(8);                             // C设置输出数据的域宽为8，只适用于它后面的第一个数据
   cout.fill('*');                            // 设置填充字符为 *
   cout.setf(ios::left, ios::adjustfield);    // 设置输出为左对齐
   cout<<x<<','<<y<<endl;
   cout.setf(ios::hex, ios::basefield);       // 设置输出数据为十六进制数
   cout<<x<<','<<y<<endl;
   cout.unsetf(ios::showbase);                // 设置输出的数据时，前面不带表示数制的前导字符
   cout<<x<<','<<y<<endl;
}
```

我们首先来看程序中A行设置的意义，从前面的分析可知:

ios::basefield的值是: 00000000 01110000

ios::oct的值是: 00000000 00100000

从表14-2中我们查得cout.setf(ios::oct,ios::basefield);的功能是: 将标志字的某些位置新值，第二个参数指定待设置的若干位，第一个参数指定这些位的新值。对于本例第二个参数

ios::basefield中有三位的值为1（从最低位开始的第5、6、7三位），其意义就是要将状态字x_flags中相应的位置的值置为新值，新值是第一个参数ios::oct中对应的三位二进制数010，可见十进制和十六进制的状态位均被置为0，只有八进制的状态被置为1，表示按八进制数输出。在x_flags中除了表示输出数制的三位被置为新值外，其他状态位不变。

程序中的B行，设置输出的数据前面带表示数制的前导字符，八进制数的前导字符为0，十六进制数的前导字符为0x。还要注意C行，设置输出数据的域宽为8，只适用于它后面的第一个数据。程序的运行结果为：

```
245, 216
0245, 0216
0245****, 0216
0xa5, 0x8e
a5, 8e
```

**例14.6** 使用成员函数，控制输入格式。

```
#include <iostream.h>
void main( )
{ char c, str[80];
  int i=0;
  cin>>c ;                    // A    在此行之前加入语句cin.unsetf(ios::skipws);
  while(c!='\n')
  {    str[i++]=c;
       cin >> c ;
  }
  str[i]='\0';
  cout << str<< endl;
}
```

此程序的运行状况如何？我们希望运行时输入：abcd<Enter>后，程序输出abcd，然后就结束。但是程序一直不停地运行下去：

```
abcd<Enter>
efg<Enter>
hij<Enter>
......
```

原因是：缺省状态下，ios::skipws值为1，表示跳过空白字符，即空白字符作为数据之间的分隔符不被提取，“<Enter>”是一种空白字符，所以'\n'永远不会被提取，变量c的值永远不会等于'\n'。解决办法是，在程序的A行之前加入语句cin.unsetf(ios::skipws);。

**例14.7** 使用成员函数，控制输入格式。

```
#include <iostream.h>
void main( )
{   char c1, c2, c3;
    cin.unsetf(ios::skipws);                    // A设置不跳过空白字符
    cin>>c1>>c2>>c3;                            // B
    cout<<c1<<c2<<c3<<'#'<<endl;
    cin.setf(ios::skipws);                      // C 设置为跳过空白字符
    cin.setf(ios::hex, ios::basefield);         // 设置为: 输入数据为十六进制数
    int x, y;
    cin>>x>>y;
    cout<<x<<','<<y<<endl;
}
```

首先，程序在A行，设置不跳过空白字符。也就是说，在做提取操作时，空白字符不再作为输入数据之间的分隔符，而是被作为一个字符提取。程序的一种运行状况如下：

```
a  b  c<Enter>      // 输入的a、b、c之间分别有一个空格
a  b#
10  20<Enter>      // 输入
12, 16
```

在做B行的提取操作后，c1、c2和c3提取得到的值分别是'a'、' '（空格）和'b'，这可以从输出上看出。此时输入缓冲区的下一个字符是字母c前的空格。接着，程序在C行，又重新设置为跳过空白字符，于是下面x和y提取的值分别是十六进制的c和10，它们对应的十进制数分别是12和16，于是输出的是十进制数12和16。输入缓冲区中余下的20就没有用了。

若将程序中的C行删除，则运行状况为：

```
a  b  c<Enter>
a b#
0, -858993460
```

产生这个输出的原因是：在正确提取了c1、c2和c3的值后，输入缓冲区中的下一个字符是空格，然后继续提取x的值，出错，于是系统将x的值置为0；下面就不再继续提取y的值了，所以y的值是随机值。这种提取数据时出现的错误并不中断程序的执行，所以程序还是执行完毕了，但提取的数据不正确。

**例14.8** 使用成员函数，控制浮点数的输出精度。

```
#include <iostream.h>
void  main( )
{  double x=12.3456789;
   int pre=cout.precision( );                    // 获得缺省精度
   cout<<"default precision="<<pre<<endl;
   cout<<x<<endl;
   cout.precision(4);                            // A 设置精度
   cout<<x<<endl;
   cout.precision(6);
   cout.setf(ios::fixed, ios::floatfield);       // B 设置以定点数格式输出
   cout<<x<<endl;
   cout.precision(2);
   cout<<x<<endl;
   cout.setf(ios::scientific, ios::floatfield); // C 设置以科学表示法格式输出
   cout<<x<<endl;
}
```

特别要注意浮点数精度的含义，在没有设置以定点数格式或以科学表示法格式输出之前，精度的含义是指输出数据的有效数字的位数。一旦设置了以定点数格式或以科学表示法格式输出之后（程序的B行之后），精度的含义就是指小数点后的数字位数。

```
default precision=6
12.3457          // 6位有效数字
12.35            // 4位有效数字
12.345679        // 小数点后6位数字
12.35            // 小数点后2位数字
1.23e+001        // 小数点后2位数字
```

### 14.3.2 使用操纵算子进行格式控制

前面介绍的使用成员函数进行格式控制的方法，每次进行格式控制时都要单独调用成员函数，而且要记住成员函数名以及一些ios类的枚举常量的意义，比较烦琐。下面我们介绍使用操纵算子

(manipulator，又译为“操作子”)进行格式控制的方法。在表14-3中列出了常用的12个操纵算子。注意若使用表14-3中最后五个带参数的操纵算子，则要包含头文件iomanip.h。

**表14-3　操纵算子的意义**

| 操纵算子 | 功　能 | 适用于输入／输出 |
|---|---|---|
| dec | 设置为十进制数 | I/O |
| hex | 设置为八进制数 | I/O |
| oct | 设置为十六进制数 | I/O |
| ws | 跳过（吃掉）输入流中的前导空白字符 | I |
| endl | 插入一个换行符（'\n'），并刷新输出流 | O |
| ends | 插入一个标志字符串结尾的空字符，适用于“字符串”流类 | O |
| flush | 刷新输出流，即将输出缓冲区中的数据全部输出到外部设备中 | O |
| setiosflags(long) | 用参数指定的标志位，设置状态字（x_flags）的相应位 | I/O |
| resetiosflags(long) | 取消参数指定的标志位的设置 | I/O |
| setfill(int) | 设置填充字符 | O |
| setprecision(int) | 设定浮点数的精度（缺省精度为6位） | O |
| setw(int) | 设置域宽（适用于紧邻其后的第一个数据项） | O |

**例14.9**　使用操纵算子控制输入／输出格式。

```
#include <iostream.h>
#include <iomanip.h>
void  main( )
{   int x, y, a, b;
    cin>>hex>>x>>y;
    cin>>oct>>a>>b;
    cout<<setw(8)<<setfill('*')<<x<<','<<setfill('$')<<setw(4)<<y<<endl;
    cout<<setiosflags(ios::left) <<a<<','<<setw(8)<<b<<endl;
}
```

注意设置域宽操纵算子setw( )仅适用于紧邻其后的第一个数据项。此程序的运行状况如下:

```
10  20<Enter>
10  20<Enter>
******16, $$32
8, 16$$$$$$
```

**例14.10**　使用操纵算子控制刷新输出流。

```
#include <iostream.h>
void  main( )
{   int x;
    clog<<"input x:"<<flush;              //A
    cin>>x;                               //B
    cout<<x<<endl;
}
```

程序的运行状况如下:

```
input x: 5<Enter>
5
```

注意本例与例14.3的区别。在A行使用操纵算子flush刷新输出流，提示信息input x:，然后输入x，输出x。或者将A行改为clog<<"input:"<<endl;，因为endl在插入一个换行符后，兼有刷新输出流的功能。

**例14.11** 使用操纵算子控制浮点数的输出精度。

```
#include <iostream.h>
#include <iomanip.h>
void  main( )
{    double x=12.3456789;
     cout<<x<<endl;
     cout<<setprecision(4)<<x<<endl;
     cout.setf(ios::fixed, ios::floatfield);          // 设置以定点数格式输出
     cout<<x<<endl;
     cout.setf(ios::scientific, ios::floatfield);     // 设置以科学表示法格式输出
     cout<<setprecision(2)<<x<<endl;
}
```

程序的运行结果如下:

```
12.3457        // 6位有效数字
12.35          // 4位有效数字
12.3457        // 小数点后4位数字
1.23e+001      // 小数点后2位数字
```

在默认情况下，输出精度为6位，表示有效数字位数。当设置使用定点数格式输出后，该精度表示输出数据小数点后的位数；当设置使用科学表示法格式输出后，该精度表示输出数据尾数部分小数点后的位数。

***例14.12** 用操纵算子控制输出格式。

```
#include <iostream.h>
#include <iomanip.h>
void main( )
{    int i, n, w;
     cout<<"Please input an odd number: ";
     cin>>n;                                          // 输入一个奇数
     for(i=0; i<n; i++)
     {     if(i<n/2) w=n/2-i;
           else w=i-n/2;
           if(i!=n/2)
                 cout<<setfill(' ')<<setw(w)<<' ';    // ' '表示空格，输出每行的前导空格
           if(i<n/2) w=i*2+1;
           else w=n-(i-n/2)*2;
           cout<<setfill('*')<<setw(w)<<'*';          // 输出每行的*号
           cout<<endl;
     }
}
```

本程序的功能是，输出一个由星号 * 组成的菱形，要求输入的n为奇数。例如，若程序输入奇数5，则输出的图形为:

```
  *
 ***
*****
 ***
  *
```

## 14.4 使用成员函数实现输入／输出

前面我们介绍通过重载运算符“>>”和“<<”实现插入和提取操作，现在我们介绍通过istream类和ostream类的成员函数实现数据的输入和输出。

### 14.4.1 输出函数

在ostream类中定义了一些公有成员函数，它们可以用来实现输出以及有关输出的一些特殊控制。常用的输出函数见表14-4。

**表14-4　控制输出的成员函数**

| 函　数 | 功　能 |
| --- | --- |
| ostream& put(char);<br>ostream& put(unsigned char);<br>ostream& put(signed char); | (1) 输出一个字符 |
| ostream& write(const char *, int);<br>ostream& write(const unsigned char *, int);<br>ostream& write(const signed char *, int); | (2) 输出指定个数的字符 |
| ostream& flush( ); | 刷新输出流 |

put函数用于输出一个字符；write函数用于输出指定个数的字符，第一个参数是输出的第一个字符在字符串中的起始地址，第二个参数指定一次输出字符的个数；flush函数用于刷新输出流，在必要时，可以用cout.flush( )刷新cout输出流，用clog.flush( )刷新clog输出流。

**例14.13** 输出成员函数的使用。

```
#include <iostream.h>
void main( )
{   int x=97, y=98;
    char c1='A', c2='B';
    cout<<c1<<c2<<char(x)<<char(y)<<endl;                          //A
    cout.put(c1).put(c2).put(char(x)).put(char(y)).put('\n');      //B
    char s[100]="programming\nlanguage\n";
    cout.write(s, 5);                                              //C
    cout.write(s+5, 5);                                            //D
    cout.write(s+10, 5);                                           //E
    cout<<endl;
}
```

注意程序中A行和B行的功能一样。C行输出5个字符progr，D行输出5个字符ammin，E行输出5个字符g\nlan（'\n'为一个字符）。程序的完整输出结果是：

```
ABab
ABab
programming
lan
```

### 14.4.2 输入函数

在istream类中定义了一些公有成员函数，它们可以用来实现输入以及有关输入的一些特殊控制。常用的输入函数见表14-5。

表14-5中前面五组函数分别用于输入字符或字符串。(1) 组函数的功能是：从输入流中提取一个字符，并将该字符作为返回值。(2) 组函数的功能也是从输入流中提取一个字符，并将该字符赋给参数字符变量。对 (1) 组和 (2) 组函数，在输入时，不跳过空白字符，即把空白字符也作为字符提取；而不像用提取运算符>>提取字符那样，在默认的情况下，把空白字符作为分隔符（不把它提取给变量）。

(3) 组和 (4) 组函数，均用于提取一行字符。提取得到的字符串存放在第一个参数指向的字符串空间中，第二个参数指定一次最多提取的字符个数，第三个参数指定行结尾标志，默认的是'\n'。如果第二

个参数的值为n，在提取时，若输入流当前行中的字符个数少于n，则提取到行结尾标志为止；若输入流当前行中的字符个数大于或等于n，则提取前n-1个字符，加上字符串结尾标志'\0'，存放到第一个参数指向的字符串空间中，输入流中余下的字符以后再提取。用户可以根据实际需要，用第三个实参指定所需的行结尾标志。(3) 组和 (4) 组函数的区别是：(3) 组函数在提取完一行字符串后，行结尾标志仍然留在了输入缓冲区中；而 (4) 组函数在提取完一行字符串后，行结尾标志已被提取出来，实际上它被丢弃，提取出的并存入内存中的字符串结尾处没有字符'\n'，而代之以'\0'。(5) 组函数的功能是，读入指定个数的字符。

**表14-5 控制输入的成员函数**

| 函 数 | 功 能 |
|---|---|
| int istream::get( ); | (1) 读入一个字符 |
| istream& istream::get(char &);<br>istream& istream::get(unsigned char &);<br>istream& istream::get(signed char &); | (2) 读入一个字符 |
| istream& istream::get(char *, int , char='\n');<br>istream& istream::get(unsigned char *, int , char='\n');<br>istream& istream::get(signed char *, int , char='\n'); | (3) 读入一行字符，不读取行结尾标志(即行尾标志依然留存在输入缓冲区中) |
| istream& istream::getline(char *, int , char='\n');<br>istream& istream::getline(unsigned char *, int , char='\n');<br>istream& istream::getline(signed char *, int , char='\n'); | (4) 读入一行字符，读取并舍弃行结尾标志 |
| istream& read(char *, int);<br>istream& read(unsigned char *, int);<br>istream& read(signed char *, int); | (5) 读入指定个数的字符 |
| istream& ignore(int =1, int =EOF); | 跳过第一个参数指定个数的字符 |
| int gcount( ); | 获得前一次读取字符的个数 |
| int peek( ); | 获得输入流中的下一个字符值，但不提取 |
| istream& putback(char); | 将一个字符退回到输入流中 |
| void eatwhite( ); | 提取（即“吃掉”）前导空白字符，仅对其后第一个输入项有效 |

**例14.14** 输入成员函数的使用。

在本例中，输入若干行字符（假定每行少于20个字符），输出每行的字符个数，最后将最长的行及其长度输出。

```
#include <iostream.h>
#include <string.h>
void main( )
{   char buf[20], max[20];
    int cnum,  mc=0;
    while(cin.getline(buf, 20))              // A提取一行字符，读取并舍弃行尾标志'\n'
    {
          cnum=cin.gcount( );                // B获取A行的getline( )实际读取的字符个数
          if(cnum>mc)
          {     mc=cnum;
                strcpy(max, buf);
          }
          cout<<"len="<<cnum-1<<","<<buf<<endl; // C
    }
    cout<<"max len="<<mc-1<<","<<max<<endl; // D
}
```

注意程序中的A行，getline函数若正确提取了数据，则返回“真”；若遇输入流结束，提取不到数据，则返回“假”。cin.getline(buf, 20)表示最多提取19个字符，加一个字符串结尾标志'\0'后存入buf。在输入时若输入Ctrl+z，则表示输入流结束，跳出A行控制的循环。注意程序中的B行，行结尾标志'\n'被统计在cnum中，所以在输出字符串长度时，需要减1，见程序中的C行和D行。程序的运行状况是：

```
an apple<Enter>
len=8, an apple
very good<Enter>
len=9, very good
pear? <Enter>
len=5, pear?
ok! <Enter>
len=3, ok!
^Z<Enter>                    // 输入Ctrl+z，表示输入流结束。
max len=9, very good
```

在例14.14中，我们使用getline函数提取一行字符，也可以使用get函数完成同样的功能。改写例14.14如下，并注意getline函数与get函数的区别。

**例14.15**　使用get函数读入一行数据的功能，实现与例14.14同样的功能。

```
#include <iostream.h>
#include <string.h>
void main( )
{   char buf[20], max[20];
    int cnum, mc=0;
    while(cin.get(buf, 20))                    // 不提取'\n'
    {     cnum=cin.gcount( );
          cin.get( );                          // 提取（“吃掉”）每行结束的'\n'
          if(cnum>mc)
          {     mc=cnum;
                strcpy(max, buf);
          }
          cout<<"len="<<cnum<<","<<buf<<endl; // cnum不需要减1
    }
    cout<<"max len="<<mc<<","<<max<<endl;
}
```

**例14.16**　使用getline函数实现特殊输入要求。

程序以时分秒的格式输入时间，即输入格式是hh:mm:ss，要求将“时分秒”分成三个字符串提取。例如：若输入12:05:16，则提取的字符串分别为"12"、"05"和"16"。

```
#include <iostream.h>
void  main( )
{   char hh[10], mm[10], ss[10];
    cin.eatwhite( );                  // 跳过输入中的前导空白字符，仅对其后的第一个输入项有效
    cin.getline(hh, 10, ':');
    cin.getline(mm, 10, ':');
    cin.getline(ss, 10, '\n');
    cout<<hh<<':'<<mm<<':'<<ss<<endl;
}
```

程序的运行状况是：

```
12:05:16<Enter>    // 前面输入若干空白字符
12:05:16
```

例14.17 使用get函数和ignore函数。

```
#include <iostream.h>
void main( )
{   char c;
    for(int i=0; i<3; i++)
    {     cin.ignore(2);              //跳过前两个字符
          do
          {     cin.get(c);
                cout.put(c);
          } while(c!='\n');
    }
}
```

程序的输出结果是:

```
milk man<Enter>
lk man
postman<Enter>
stman
a student<Enter>
student
```

## 14.5 重载插入和提取运算符

通过前面的学习，我们知道在istream类和ostream 类中，已用成员函数实现了重载的提取运算符>>和插入运算符<<，实现标准数据类型数据的输入和输出。如: `int x; cin>>x;`，则编译器自动解释成`cin.operator>>(x);`。对于用户新定义的数据类型如复数Complex类、日期Date类和时间Time类等，能否直接使用>>和<<进行提取和插入操作呢？即如果定义了两个复数类对象`Complex c1,c2;`，能否实现复数对象的整体输入／输出，也就是说，是否可以 `cin>>c1;`或`cout << c2;`？回答是肯定的，但必须首先定义运算符重载函数、重载运算符>>和<<。

标准数据类型的提取运算符（>>）和插入运算符（<<）是通过重载函数来实现的，它们是istream类和ostream 类的成员函数。对于用户定义的类，也可以重载提取运算符（>>）和插入运算符（<<），但不能把重载函数定义成istream类和ostream 类的成员函数或友元函数。因为istream类和ostream类是系统定义好的，用户不应修改它。

一般来说，可以把用户定义类的对象的整体插入和提取操作的运算符重载为类的友元函数，请看例14.18。

例14.18 用友元函数实现复数类对象插入和提取运算符重载函数。

```
#include<iostream.h>
class Complex
{
    double Real, Image;
public:
    Complex(double r=0, double i=0)
    {   Real=r; Image=i;}
    friend istream& operator>>(istream &, Complex &);   //友元
    friend ostream& operator<<(ostream &, Complex &);   //友元
};
istream& operator>>(istream &in, Complex &c)
{   in>>c.Real>>c.Image;
    return in;
```

```
}
ostream& operator<<(ostream &out, Complex &c)
{   out<<c.Real;
    if(c.Image>0) out<<'+'<<c.Image<<'i';                    // 如果是正数，应显式输出正号
    else if(c.Image<0) out<<c.Image<<'i';                   // 如果是负数，自动输出负号
    out<<endl;
    return out;
}
void main( )
{   Complex c1(1, 2), c2;
    cout<<c1;
    cout<<"Please input c1 & c2:\n";
    cin>>c1>>c2;                                            //A
    cout<<c1<<c2;                                           //B
}
```

程序的运行状况如下:

```
1+2i
Please input c1 & c2:
5 8<Enter>                          // 输入
7 -3<Enter>                         // 输入
5+8i
7-3i
```

对本例而言，插入和提取操作运算符重载函数只能定义成Complex类的友元函数，而不能定义成Complex类的成员函数，因为第一个参数是非Complex类的对象。

定义友元函数的目的是可以在友元函数中直接访问类的私有成员，但私有成员的访问可以通过公有函数接口实现。因此，也可以将插入和提取操作运算符重载函数定义成非友元方式，请看例14.19。

**例14.19**　用非友元函数实现复数类对象插入和提取操作运算符重载函数。

```
#include<iostream.h>
class Complex
{
    double Real, Image;
public:
    Complex(double r=0, double i=0)
    {   Real=r;Image=i;}
    void SetReal(double r)
    {   Real=r;}
    void SetImage(double r)
    {   Image=r;}
    double GetReal( )
    {   return Real;}
    double GetImage( )
    {   return Image; }
};
istream& operator>>(istream &in, Complex &c)
{   double r, i;
    in>>r>>i;
    c.SetReal(r);
    c.SetImage(i);
    return in;
```

```
}
ostream& operator<<(ostream &out, Complex &c)
{   double r, i;
    r=c.GetReal( ); i=c.GetImage( );
    out<<r;
    if(i>0) out<<'+'<<i<<'i';                          // 如果是正数，应显式输出正号
    else if(i<0) out<<i<<'i';                          // 如果是负数，自动输出负号
    out<<endl;
    return out;
}
void main( )
{   Complex c1(1, 2), c2;
    cout<<c1;
    cout<<"Please input c1 & c2:\n";
    cin>>c1>>c2;
    cout<<c1<<c2;
}
```

本例用非友元非成员的方法实现了插入和提取运算符重载，本例的效果与例14.18一样。但是一般来说不提倡这种做法，因为通过公有成员函数访问私有成员，降低了运行效率。

**例14.20** 用非友元函数实现日期类对象插入和提取操作运算符重载函数。

本例中定义了一个日期类Date，它的三个数据成员分别表示年、月、日。要求定义重载函数实现日期对象的插入和提取工作。要求输入格式为“yyyy/mm/dd”，其中yyyy表示年，最多4位数；mm和dd分别表示月和日，最多两位数。输出格式为“yyyy/mm/dd”，年、月、日固定为4位、2位和2位。例如输入2003/8/1，则输出2003/08/01。注意，本程序对输入日期的合法性未做检查，感兴趣的读者可以自行编制函数检查日期的合法性。

```
#include <iostream.h>
#include <string.h>
class  Date
{
    int year, month, day;                              // 年、月、日
    void   itoa(int n, char *s, int fixlen)            // A
    {      for(int i=0; n; n=n/10, i++)
                s[i]=n%10+'0';
           for(; i<fixlen; i++)
                s[i]='0';
           s[i]='\0';
           int j, t;
           for(j=0; j<i/2; j++)                        // 逆置
                { t=s[j]; s[j]=s[i-1-j]; s[i-1-j]=t; }
    }
    int atoi(char *s)                                  // B
    {      int n=0;
           for( ;*s; s++)    n=n*10+*s-'0';
           return(n);
    }
public:
    Date(int y=0, int m=0, int d=0)
    {     year=y;
          month=m;
```

```
        day=d;
    }
    void InputDate(istream &in)
    {   char y[10], m[10], d[10];
        in.getline(y, 10, '/');                    //C
        in.getline(m, 10, '/');                    //D
        in.getline(d, 10, '\n');                   //E
        year=atoi(y);
        month=atoi(m);
        day=atoi(d);
    }
    void ShowDate(ostream &out)
    {   char DateStr[20];
        char temp[10];
        itoa(year, DateStr, 4);
        itoa(month, temp, 2);
        strcat(DateStr, "/");
        strcat(DateStr, temp);
        itoa(day, temp, 2);
        strcat(DateStr, "/");
        strcat(DateStr, temp);
        out<<DateStr;
    }
};
istream& operator>>(istream &in, Date &d)          //F
{   d.InputDate(in);
    return in;
}
ostream& operator<<(ostream &out, Date &d)         //G
{   d.ShowDate(out);
    return out;
}
void main( )
{   Date d1(1999, 3, 8), d2(2003, 5, 16);
    cout<<d1<<endl<<d2<<endl;
    cin>>d1;
    cout<<d1<<endl;
}
```

本程序的一次运行结果如下:

```
1999/03/08
2003/05/16
2005/7/7<Enter>   // 输入
2005/07/07
```

本程序中A行定义了一个私有成员函数void itoa(int n, char *s, int fixlen)，它的功能是将整数n转换成固定长度fixlen的字符串，存放在s指向的空间中，若n不足fixlen位，则前面补0。例如: char temp[20];，若有调用itoa(2003, temp, 4)，则结果temp中的字符串为"2003"；若有调用itoa(8, temp, 4)，则结果temp中的字符串为"0008"。该函数可以用一个更简单的方式实现:

```
void  itoa(int n, char *s, int fixlen)
{     s[fixlen]='\0';
```

```
    for(int i=fixlen-1; n>0; i--, n/=10)        s[i]=n%10+'0';
    for( ; i>=0; i-- ) s[i]='0';
}
```

本程序中B行定义了一个私有成员函数int atoi(char *s)，它的功能是将参数指向的字符串转换成整数，作为函数的返回值。例如: 若有int y; char s[20]="1999";，当调用y=atoi(s);后，y得到整数值1999。

程序中C行和D行，提取的一行以'/'结尾。E行提取的一行以'\n'结尾。这三行用于分别提取输入日期2005/7/7中的年月日三个部分。

程序中F行和G行，定义了Date类的插入和提取运算符重载函数。

## 14.6 文件流类

输入／输出外部设备可以是具体的物理设备如键盘、显示器等，也可以是一个磁盘文件。键盘和显示器是标准输入／输出文件，它们的输入／输出控制在前面已进行了较详细的描述。本节主要介绍对磁盘文件的读写。

磁盘文件是一组信息的集合，它被存放在磁盘上。我们用文件名标识它，它所包含的信息是文件的内容。磁盘文件分为文本文件和二进制文件。在文本文件中存放的是“文本流”，即表示信息的一串ASCII字符。在二进制文件中存放的是“二进制流”，是信息的内存映像，按信息在内存中的二进制格式存放。在“流”中的数据是以字节为单位的。

C++中专门定义文件流类，用于处理文件的读写操作。

### 14.6.1 文件流类体系

基本文件流类体系如图14-6所示。

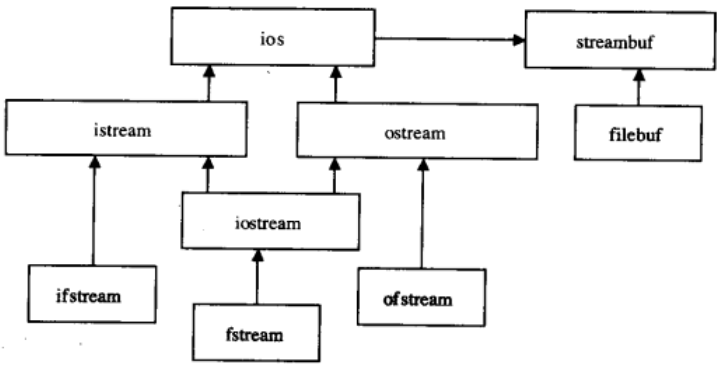



图14-6 基本文件流类体系

在文件流类体系中， ios是所有输入／输出流类的基类，其他的类均是从该类派生出来的，它是在头文件“ios.h”中定义的。类filebuf管理文件的输入／输出缓冲区，用户的应用程序中一般不涉及该类。注意最下面的三个派生类， ifstream类用于管理文件的输入，即从文件中提取数据。ofstream类用于管理文件的输出，即向文件插入数据。fstream类用于管理文件的输入和输出，即完成文件中数据的提取和插入操作。

### 14.6.2 文件的打开和关闭

使用文件流类，必须包含头文件fstream.h。文件的使用过程如下:

(1) 定义一个文件流对象。

(2) 打开文件。

(3) 对文件进行读写操作。

(4) 关闭文件。

下面分别做详细介绍。

### 1. 定义一个文件流对象

```
如: ifstream  infile;     //A
    ofstream  outfile;    //B
    fstream   iofile;     //C
```

A行定义了一个输入文件流对象infile，用它与一个输入文件建立联系。B行定义了一个输出文件流对象outfile，用它与一个输出文件建立联系。C行定义了一个输入／输出文件流对象iofile，用它与一个输入／输出文件建立联系。输入／输出文件是指，既可从该文件中提取信息，又可向该文件插入信息。

那么具体对哪一个文件进行读写呢？这就是“打开文件”完成的工作。

### 2. 打开文件

**• 使用成员函数open打开文件**

可通过已定义的文件流对象，调用成员函数，打开文件，与具体待读写的文件建立联系。三个文件流类ifstream、ofstream和fstream各自定义了自己的打开文件成员函数（具体的定义可参见标准头文件fstream.h），它们是：

```
void  ifstream::open(const char *, int=ios::in, int=filebuf::openprot);
void  ofstream::open(const char *, int=ios::out, int=filebuf::openprot);
void  fstream::open(const char *,  int,  int=filebuf::openprot);
```

第一个参数是待打开的文件名，用字符串表示，文件名可以是带路径的文件名。如"data.txt"表示当前工作目录下的文件data.txt，"C:\\prog\\data.txt"表示C盘prog目录下的文件data.txt。注意字符串中的双反斜杠表示一个'\'字符，因为反斜杠是转义字符的起始字符，所以规定用双反斜杠表示一个'\'字符。

第二个参数是文件的打开方式，前两个函数该参数的缺省值分别是ios::in和ios::out，分别表示输入和输出；第三个函数的该参数无缺省值，表示调用该函数时，必须指定输入／输出方式。

第二个参数的完整定义在头文件ios.h中。在ios类中定义的公有枚举类型open_mode，其枚举常量的意义如下：

```
enum open_mode
{   in          = 0x01,          //输入（读）方式
    out         = 0x02,          //输出（写）方式
    ate         = 0x04,          //文件指针位于文件尾
    app         = 0x08,          //输出追加（增补）方式
    trunc       = 0x10,          //截断方式
    nocreate    = 0x20,          //不创建新文件
    noreplace   = 0x40,          //不替换方式
    binary      = 0x80           //二进制方式
};
```

引用枚举成员的方式为：ios::in、ios::out等，其他的可以类推。每一个枚举成员表示打开状态的一个二进制位，可以用位运算的或“|”来组合各种状态。

以上文件的打开方式，有些可以单独使用，有些必须组合使用。常用的文件打开方式（及组合）有：

```
ios::in                              //以读方式打开正文文件
ios::in|ios::out                     //以读写方式打开正文文件
ios::in|ios::binary                  //以读方式打开二进制文件
ios::in|ios::out|ios::binary         //以读写方式打开二进制文件
ios::in|ios::nocreate                //以读方式打开正文文件，若正文文件不存在，则打开不成功
ios::in|ios::binary|ios::nocreate    //以读方式打开二进制文件，若文件不存在，则打开不成功
ios::out                             //以写方式打开正文文件
ios::out|ios::binary                 //以写方式打开二进制文件
```

```
ios::out|ios::app                    //以追加写方式打开正文文件
ios::out|ios::binary|ios::app        //以追加写方式打开二进制文件
ios::out |ios:: noreplace            //以写方式打开正文文件，若正文文件存在，则打开不成功
ios::out |ios::binary|ios:: noreplace  //以写方式打开二进制文件，若文件存在，则打开不成功
```

第三个参数是文件打开时的保护方式，缺省值是filebuf::openprot，表示“可共享的”，即该文件可同时被多个文件流对象打开。第三个参数的取值及意义如下：

```
filebuf::openprot                    //缺省共享模式
filebuf::sh_none                     //不允许共享（独占方式）
filebuf::sh_read                     //只允许读共享
filebuf::sh_write                    //只允许写共享
```

下面举三个打开文件的例子：

```
ifstream  infile;
infile.open("text.dat");
```

表示打开数据文件text.dat，作为输入文件。

```
ifstream  infile;
infile.open("text.dat", ios::in, filebuf::sh_none);
```

表示以独占方式打开数据文件text.dat，作为输入文件。打开方式为独占方式的意义是：在输入流对象infile打开文件text.dat的同时，不允许其他输入／输出流对象同时打开text.dat文件。其目的是保护文件内容，在读该文件的同时，不允许对该文件进行其他操作。

```
ofstream  outfile;
outfile.open("text.out");
```

表示打开数据文件text.out，作为输出文件。

**• 使用类的构造函数打开文件**

除了可以使用成员函数open打开文件，也可以使用三个文件流类ifstream、ofstream和fstream的构造函数打开文件。这三个类各自定义了自己的构造函数，用于在定义文件流对象的同时，打开文件。这些构造函数是：

```
ifstream::ifstream(const char *, int =ios::in, int = filebuf::openprot);
ofstream::ofstream(const char *, int =ios::out, int = filebuf::openprot);
fstream::fstream(const char *, int, int = filebuf::openprot);
```

我们注意到，这三个重载函数的参数与open函数一致，其意义也是一致的。如上面两个打开文件的例子，也可以用构造函数打开文件：

```
ifstream infile;
infile.open("text.dat");
```

等价

```
ifstream infile("text.dat");
```

```
ifstream infile;
infile.open("text.dat", ios::in, filebuf::sh_none);
```

等价

```
ifstream infile("text.dat", ios::in, filebuf::sh_none);
```

### 3. 对文件进行读写操作

对文件进行读写操作所使用的提取和插入运算符、读写函数等，与前面介绍的标准输入／输出设备的读写是一样的。因为不论是标准输入／输出流对象cin、cout，还是本节介绍的文件输入／输出流对象，它们都是istream和ostream类的派生类的对象，都继承了istream和ostream类的重载的插入和提取运算符，同时也继承了输入／输出成员函数。我们将在后面的例子中看到具体的输入／输出的用法。

在对文件进行读写时，我们将文件看成字符流，有一个读写位置指针从字符流中扫过。读写位置指

针就是进行提取和插入操作的当前点，如图14-7所示。

文件打开时，读写位置指针在文件的首部，当进行提取或插入操作时，读写位置指针自动向后移动。

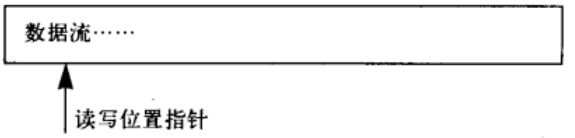



图14-7　读写位置指针示意

#### 4. 关闭文件

当打开一个文件时，系统要为文件的使用分配一定的系统资源，如缓冲区等。因此一般来说，当对一个文件读写完毕时，要关闭该文件，以释放所占用的系统资源。虽然在程序运行结束时，系统会自动调用文件流对象的析构函数关闭文件，但一般系统对同时可打开的文件数是有限制的，所以及时关闭已使用完毕的文件是一个好的习惯。另外一个好处是，关闭一个文件后，还可以将与这个文件关联的文件流对象，关联到其他文件上。

与打开文件对应，三个文件流类ifstream、ofstream和fstream均有关闭文件的成员函数：

```
void ifstream::close( );
void ofstream::close( );
void fstream::close( );
```

如：对上述已打开的三个文件流对象infile、outfile和iofile，可以调用成员函数关闭它们：

```
infile.close( );
outfile.close( );
iofile.close( );
```

### 14.6.3　文本文件的读写

**例14.21**　编写一个程序将数值1~100及每个数的平方根写入文件sqrttable.txt。

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>
void main( )
{   double x;
    ofstream out;                          // 定义一个输出流对象
    out.open("sqrttable.txt");             // 打开文本文件sqrttable.txt
    for(x=1; x<=100; x++)
        out<<x<<'\t'<<sqrt(x)<<endl;       // 将结果写入文件
    out.close( );                          // 关闭文件
}
```

因涉及文件操作，故在程序的开头必须包含头文件fstream.h。本程序运行结束后，读者可以打开文本文件sqrttable.txt观察结果。

**例14.22**　编写一个程序，用于显示一个文本文件。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void main( )
{   char filename[40], line[80];
    ifstream  infile;
    cout<<"Please input a filename: ";
    cin>>filename;                                  //A
    infile.open(filename);                          //B
    if(!infile)                                     //C
```

```
    {       cout<<"Can not open file: "<<filename<<endl;
            exit(1);                                    //D
    }
    while(infile.getline(line, 80))                     //E
          cout<<line<<endl;
    infile.close( );                                    //F
}
```

程序在A行输入一个文本文件名，在B行打开该文件，注意ifstream类的对象infile调用成员函数open，默认的打开方式为ios::in，所以我们缺省第二个参数。如果打开成功，则文件流对象infile的值为非0值；若打开不成功，则infile的值为0。所以在C行根据infile的值判断打开是否成功，若打开不成功，则在D行用exit函数结束程序的运行，exit函数的参数为返回码，操作系统可以根据该返回码做相应的处理，一般编程者不必关心。如果使用exit函数，则在程序的开头必须包含头文件stdlib.h，因为在该头文件中对exit函数做了说明。程序在E行通过对象infile调用成员函数getline输入一行信息，getline函数的功能与前面介绍的通过标准输入流cin调用getline函数的功能一样。getline函数如果提取成功，则返回非0值；若提取不成功（当遇到文件结束符时，提取不成功），则返回0值；程序的处理过程是，从文本文件中读入一行，向标准输出设备输出一行，程序的功能就是将一个文本文件的内容显示在屏幕上。最后在F行，关闭该文件。

运行该程序时，我们可以输入任意一个**已存在的**文本文件的文件名，如一个cpp源程序名（扩展名为.cpp的程序），或一个记事本文件名（扩展名为.txt的文件）。假如输入了一个文件名，对应的文件不存在，则如果用B行的打开方式，系统会自动建立一个以输入的文件名为名字的空文件，也就是说打开函数的执行还是成功的，不会输出信息Can not open file: filename。因为是空文件，所以后面什么也不输出。如果我们希望在B行打开文件时，若文件存在，则打开成功；若不存在，则打开不成功，则应将B行改为：

```
infile.open(filename, ios::in|ios::nocreate);     //B
```

在例14.22中，我们用读入一行显示一行的方式编写程序。下面我们用读入一个字符显示一个字符的方式来实现显示正文文件的功能。

**例14.23** 编写一个程序用于显示一个正文文件。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void  main( )
{   char filename[40], ch;
    fstream  infile;                                    //A
    cout<<"Please input a filename: ";
    cin>>filename;
    infile.open(filename, ios::in|ios::nocreate);       //不能缺省第二个参数
    if(!infile)
    {       cout<<"Can not open file: "<<filename<<endl;
            exit(1);
    }
    infile.unsetf(ios::skipws);                         //B
    while(infile>>ch)                                   //C
          cout<<ch;
    infile.close( );
}
```

注意程序A行，我们使用的是fstream类的对象，完成输入操作，下面调用open函数时，第二个参数不能缺省。B行设置提取状态为，提取时不跳过空白字符，即空白字符也作为一个字符被提取出来。C行

用重载的提取运算符进行提取操作，若提取成功，则返回非0值；若提取不成功，则返回0值。

因为get在做提取操作时不跳过空白字符，因此如果使用成员函数get进行提取操作，则程序中方框括起来的部分可改为：

```
while(infile.get(ch))                    //C
        cout<<ch;
```

**例14.24** 编写一个程序用于复制文本文件。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void  main( )
{
    char infilename[40], outfilename[40], ch;                    //A
    cout<<"Please input an input file name: ";
    cin>>infilename;
    fstream  infile(infilename, ios::in|ios::nocreate);          //B
    if(!infile)
    {
          cout<<"Can not open input file: "<<infilename<<endl;
          exit(1);
    }
    cout<<"Please input an output filename: ";
    cin>>outfilename;
    fstream  outfile(outfilename, ios::out);                     //C
    if(!outfile)
    {
          cout<<"Can not open output file: "<<outfilename<<endl;
          exit(2);
    }
    while(infile.get(ch))
         outfile<<ch;                                            //D
    infile.close( );
    outfile.close( );
}
```

程序在A行定义了两个字符数组infilename和outfilename分别用于存放源文件名和目标文件名。程序在B行和C行使用构造函数分别打开输入流对象infile和输出流对象outfile，注意它们都是fstream 类的对象，B行和C行的第二个参数都不能缺省。在D行使用重载的插入运算符对目标文件进行插入操作。

在C行用ios::out方式打开文件，若目标文件不存在，则系统创建一个新文件作为目标文件；若目标文件已存在，则系统首先删除该文件，然后再以相同的文件名创建一个新目标文件。所以为了防止一个已存在的文件被误删除，可以将C行改为：

```
fstream  outfile(outfilename, ios::out|ios::noreplace);          //C
```

此时，若目标文件已存在，则打开不成功。

读者可以用此程序复制一个文本文件，然后用例14.22的程序验证复制的目标文件和源文件的内容是否一样。

如果将例14.24的B行和C行分别改为：

```
fstream  infile(infilename, ios::in|ios::binary|ios::nocreate);  //B
fstream  outfile(outfilename, ios::out|ios::binary);             //C
```

即以二进制方式打开文件，则本程序可用于复制任意类型的文件。请读者自行验证程序的正确性。

如复制图像文件（如扩展名为.bmp、.jpg的文件）、音频文件（如扩展名为.rm、.mp3的文件）、可执行文件（如扩展名为.exe、.com的文件）、Word文档（如扩展名为.doc、.rtf的文件）等。

**例14.25**　编写一个程序从一个文本文件source.txt中读入若干整数，用选择法将这些数据排成升序，将排序后的结果写入另一个文本文件target.txt中。注意两个文件均在d盘的data文件夹中。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void sort(int *a, int n)
{   int i, j, p, t;
    for(i=0; i<n-1; i++)
    {     p=i;
          for(j=i+1; j<n; j++)
                if(a[j]<a[p]) p=j;
          if(p!=i)
          { t=a[i];a[i]=a[p]; a[p]=t; }
    }
}
void main( )
{   int a[100], i, n;
    fstream in, out;
    in.open("d:\\data\\source.txt", ios::in|ios::nocreate);   //A
    if(!in)
    {    cout<<"Can not open source.txt!"<<endl;    exit(1);  }
    out.open("d:\\data\\target.txt", ios::out);          //B
    if(!out)
    {    cout<<"Can not open target.txt!"<<endl;exit(2);  }
    i=0;
    while(in>>a[i]) i++;                                  //循环结束后，i是整数的个数
    n = i ;
    sort(a, n);
    for(i=0; i<n; i++)out<<a[i]<<endl;
    in.close( );
    out.close( );
}
```

在程序运行前，先准备好输入数据文件source.txt，放入d盘的data文件夹中，内容可以如下（其中的数值以及每行的数据个数可以任意输入）：

```
2    3    10    45    33    8    9    20    45    67    888    3
7    2    32    -2    0     -1
```

程序运行结束后，查看d盘的data文件夹中的结果文件target.txt内容是否按升序排列。

***例14.26**　编写一个程序完成一种编译预处理工作，将一个cpp源程序中的注释语句删除。在程序中，注释有两种形式。第一种形式为每行的“//”后的内容为注释；第二种形式为在“/*”和“*/”之间的内容是注释。注释语句对产生目标文件没有用处，所以一般在编译预处理阶段，将注释语句删除，然后对删除注释后的源程序进行编译（当然还要做其他的编译预处理工作）。在这里我们假定，注释不出现在字符串常数内，而且注释不能嵌套使用。

例如源程序为t.cpp，删除注释语句后产生的新的源程序为t1.cpp，它们的内容如下：

t.cpp

```
/ ******************************
*   思考一下此程序的功能?          *
****************************** /
#include <iostream.h>
#include <string.h>
void swap(char s1[], char s2[])
{    char t;
     t=*s1; *s1=*s2; *s2=t;
}
void main()
{
    char s1 []="BD", s2[]="AB";
    if(strcmp(s1,s2)>0) /*comment */
          swap(s1,s2);
    cout<<s1 <<','; // comment1
    cout<<s2<<endl;     //  comment2
}
```

t1.cpp

```
#include <iostream.h>
#include <string.h>
void swap(char s1[] char s2[])
{
     char t;
     t=*s1; *s1=*s2; *s2=t;
}
void main()
{
      char s1 []="BD", s2[]="AB";
      if(strcmp(s1,s2)>0)
            swap(s1,s2);
      cout<<s1 <<',';
      cout<<s2<<endl;
}
```

在编写程序时，我们将源程序文件看成字符流，读写位置指针从字符流中扫过，分两种状态，当文件指针在正常语句位置时，为状态1；当读写位置指针在“/*”和“*/”之间的注释位置时，为状态2。**若当前为状态1**，则：(1) 若当前字符为‘/’，其后一个字符为‘*’，则进入状态2。(2) 若当前字符为‘/’，其后一个字符为‘/’，则将本行其后的所有字符读出来（'\n' 不读出），但不写到结果文件中，仍然保持状态1。(3) 否则，将当前字符写入结果文件，仍然保持状态1。**若当前为状态2**，则如果当前字符为‘*’，其后一个字符为‘/’，则读入一个字符后，返回状态1。程序如下：

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void  main( )
{   char ch1, ch2, temp[100];
    int state=1;
    fstream  infile("t.cpp", ios::in|ios::nocreate);
    if(!infile)
    {      cout<<"Can not open source file:t.cpp"<<endl;
           exit(1);
    }
    fstream  outfile("t1.cpp", ios::out);
    if(!outfile)
    {       cout<<"Can not open target file: t1.cpp"<<endl;
            exit(2);
    }
    while(infile.get(ch1))
    {     ch2 = infile.peek( );                          //A
          if(state==1)
          {      if(ch1=='/'&&ch2=='*')
                 {      state=2; infile.get(ch2); }
                 else if(ch1=='/'&&ch2=='/')
                        infile.get(temp, sizeof(temp));       //B
                 else
                        outfile.put(ch1);
          }
```

```
            else if(ch1=='*'&&ch2=='/')
            {       infile.get(ch1);
                    state=1;
            }
        }
        infile.close( );
        outfile.close( );
    }
```

程序中A行peek( )函数的功能是取得输入流中的下一个字符的值，但不提取。程序中B行的功能是：将当前行注释忽略，即读入源程序当前行余下的内容，但不读取'\n'。注意此程序对一些特殊情况未做考虑（如嵌套注释的情况，字符串中出现双反斜杠的情况）。如读者有兴趣，可以考虑这些特殊情况，然后对程序做进一步改进。

请读者思考一下程序 t.cpp 的功能，它的正确输出为：AD，BB。

### 14.6.4 二进制文件的读写

在打开二进制文件时，打开方式中要加入ios::binary。

我们用ostream类的成员函数write( )向二进制文件中写数据，用istream类的成员函数read( )从二进制文件中读数据。这两个成员函数的功能说明在表14-4和表14-5中，这里我们只列出两个原型，供读者再次了解它们的功能：

```
ostream& write(const char *, int);
istream& read(char *, int);
```

第一个参数指定内存中某个地址，从该地址开始进行数据的读写；第二个参数指定一次读写的字节数。若用write( )函数，则表示将内存中指定地址开始的指定个数的字节的内容写入到文件中。若用read( )函数，则表示将文件中指定个数的字节的数据读入并存放在内存中指定地址开始处。

**例14.27** 编写一个程序对二进制文件进行读写。本程序的功能是，从键盘输入若干学生的信息，写入二进制文件，再从该二进制文件中读出学生的信息，输出到屏幕上。

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
struct student                                          //定义一个结构体类型
{
    char name[10];                                      //姓名
    char id[10];                                        //学号
    int score;                                          //分数
};
#define LEN  sizeof(struct student)
void  main( )
{   student st;
    fstream file("stud.dat", ios::out|ios::binary);     //以二进制方式打开输出文件
    if(!file)
    {       cout<<"Can not open output file: stud.dat"<<endl;
            exit(1);
    }
    cin>>st.name;
    while(strcmp(st.name, "#")!=0)                      //循环输入时，以输入姓名为“#”结束
    {     cin>>st.id>>st.score;                         //循环从键盘输入学生信息
```

```
            file.write((char *)&st, LEN);          // 一次写出LEN字节的内存数据
            cin>>st.name;
        }
        file.close( );                             // 关闭与file关联的文件，以便后面重复使用
                                                      file对象
        student sts[100];
        int i=0, j;
        file.open("stud.dat", ios::in|ios::binary|ios::nocreate);     // 重复使用file对象
        if(!file)
        {       cout<<"Can not open input file: stud.dat"<<endl;
                exit(2);
        }
        while(file.read((char *)(sts+i), LEN))     // 一次读入LEN字节的数据，存入内存指定地址
            i++;
        for(j=0; j<i; j++)                         // 循环向屏幕输出学生信息
              cout<<sts[j].name<<'\t'<<sts[j].id<<'\t'<<sts[j].score<<endl;
        file.close( );
    }
```

程序的一次执行状况如下:

```
wss 0101 80<Enter>
tyy 0102 90<Enter>
czz 0103 85<Enter>
#<Enter>
wss     0101    80
tyy     0102    90
czz     0103    85
```

**例14.28**　二进制文件的读写。

已知一个文本文件data.txt，其中已存放了按升序排列的若干整数，编一个程序将其中的整数读出并写到二进制文件data.bin中。二进制文件data.bin中的第一个数必须是其后的整数的个数。假定文本文件data.txt的内容如下:

```
1  2  4  6  9  10  11  15  17  21  22  23  25  26  28  29  30  32
```

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void  main( )
{   int a[100], n=0;
    ifstream  infile("data.txt", ios::in|ios::nocreate);
    if(!infile)
    {       cout<<"Can not open input file: data.txt"<<endl;
            exit(1);
    }
    while(!infile.eof( ))                                    //A
          infile>>a[n++];
    infile.close( );
    ofstream  outfile("data.bin", ios::out|ios::binary);
    if(!outfile)
    {     cout<<"Can not open output file: data.dat"<<endl;
          exit(2);
```

```
    }
    outfile.write((char *)&n, sizeof(int));                    //B
    outfile.write((char *)a, n*sizeof(int));                   //C
    outfile.close( );
}
```

程序中A行的eof( )函数是ios类的成员函数，用于判断文件流类对象的读写位置指针是否到达了文件结尾。当到达文件结尾时，eof( )函数的返回值为“真”（即非0值）；当没有到达文件结尾时，eof( )函数的返回值为“假”（0值）。程序中的B行，将整数的个数写到结果文件中。程序中的C行，将内存中从数组a的起始地址开始的连续的n个整数的存储空间的内容，一次写入二进制结果文件中。

请读者编写一个程序，将本例的结果文件data.bin中的内容读出，显示到屏幕上，以验证文件data.bin的正确性。

## 14.7 文件的随机访问

我们在图14-7中看到，文件刚被打开时，读写位置指针在文件的首部，当进行提取或插入操作时，读写位置指针自动向后移动。这称为文件的顺序读写（顺序访问）。有时，我们只需要读取文件中的部分内容，这些内容可能不在文件的首部，此时我们可以将文件读写指针移到待读写的位置，进行读写，这就是文件的随机读写（随机访问）。

istream类和ostream类提供了移动读写位置指针的成员函数，通过它们控制读写位置指针的移动，实现文件的随机读写。这些成员函数的原型是：

```
istream & istream::seekg(streampos);                           //A
istream & istream::seekg(streamoff, ios::seek_dir);            //B
streampos & istream::tellg( );                                 //C
ostream & ostream::seekp(streampos);
ostream & ostream::seekp(streamoff, ios::seek_dir);
streampos & ostream::tellp( );
```

前三个函数，函数名以g结尾（g是英文单词get的第一个字符），适用于输入流操作。后三个函数，函数名以p结尾（p是英文单词put的第一个字符），适用于输出流操作。

streampos和streamoff是数据类型，在头文件iostream.h 或istream.h或ostream.h中均对它们做了定义：

```
typedef long streamoff ;      // off表示offset，相对位置
typedef long streampos;       // pos表示position，绝对位置
```

可以看出，本质上，streampos和streamoff就是long型类型，把它们写成新的数据类型名，是为了强调上面一组函数的参数或返回值分别表示绝对位置和相对位置。

另外，在 ios 类中定义公有的枚举类型：

```
enum  seek_dir
{
    beg=0,    // 表示从文件开始处，枚举常量为: ios::beg
    cur=1,    // 表示从当前位置处，枚举常量为: ios::cur
    end=2     // 表示从文件结束处，枚举常量为: ios::end
};
```

我们用移动输入流读写位置指针的三个函数为例，来说明它们的功能。A行的seekg成员函数的功能是，将读写位置指针移动到参数指定的字节处，这是指针的绝对移动，即从文件起始位置开始移动到参数指定的字节处。B行的seekg成员函数的功能是，将第二个参数作为指针移动的参照点，将指针相对移动若干字节，第一个参数指定移动的字节数。例如infile.seekg(100, ios::cur)，表示将输入流对象infile的读写位置指针从当前位置开始向输入流结束方向移动100个字节。而infile.seekg(-100, ios::cur)，表示将输入

流对象infile的读写位置指针从当前位置开始向输入流开始方向移动100个字节。B行的seekg成员函数实现指针的相对移动。C行的tellg函数功能是：返回指针的当前位置（即相对于文件起始位置的字节数）。

我们可以从输入流的三个函数的功能，类推出输出流成员函数seekp和tellp的功能。

**例14.29**　文件的随机访问（一）。

编写一个程序，将Fibonnaci数列的前40项写入二进制文件fib.bin中，然后输出其中的奇数项，每行输出4个数。该数列的第1项和第2项均为1，从第3项开始，每一项的值是其紧邻的前两项之和，数列为：1, 1, 2, 3, 5, 8, 13, 21, 34……程序如下：

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <iomanip.h>
void  main( )
{   fstream  iofile("fib.bin", ios::in|ios::out|ios::binary);          //A
    if(!iofile)
    {     cout<<"Can not open file: fib.bin"<<endl;
          exit(1);
    }
    streampos pos = iofile.tellp( );                                   //B
    cout<<"Begin posi="<<pos<<endl;
    int f1=1, f2=1;
    for(int i=0; i<20; i++)
    {     iofile.write((char*)&f1, sizeof(int));
          iofile.write((char*)&f2, sizeof(int));
          f1=f1+f2;
          f2=f2+f1;
    }
    pos=iofile.tellp( );                                               //C
    cout<<"End posi="<<pos<<endl;
    iofile.seekg(0);                                                   //D
    for(i=0; i<20; i++)
    {     iofile.read((char*)&f1, sizeof(int));                        //E
          iofile.seekg(sizeof(int), ios::cur);                         //F
          cout<<setw(10)<<f1;
          if((i+1)%4==0) cout<<endl;
    }
    iofile.close( );
}
```

程序中的A行打开一个输入／输出二进制文件。程序中的B行定义了一个streampos类型的变量pos，并调用成员函数获得文件刚打开时，文件读写指针的位置。当向文件中写完40个数后，在程序中的C行获得当前文件读写指针的位置。程序中的D行在不关闭文件的情况下，将读写位置指针重新定位于文件起始位置。程序中的E行读入奇数项数据，程序中的F行跳过偶数项的数据。程序的运行结果如下：

```
Begin posi=0
End posi=160
         1         2         5        13
        34        89       233       610
      1597      4181     10946     28657
     75025    196418    514229   1346269
   3524578   9227465  24157817  63245986
```

**例14.30** 文件的随机访问（二）。

在例14.28中产生了一个非递减序列的二进制文件data.bin，文件中的第一个数是序列中的数据个数。编写程序，在不读出文件中全部数据的情况下，用二分法查找任意一个数是否存在于序列中，如果存在，则指出它在序列中的序号。约定序列的序号从1开始。序列如下：

| 1 | 2 | 4 | 6 | 9 | 10 | 11 | 15 | 17 | 21 | 22 | 23 | 25 | 26 | 28 | 29 | 30 | 32 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

程序如下：

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void  main( )
{    fstream  infile("data.bin", ios::in|ios::binary|ios::nocreate);
     if(!infile)
     {       cout<<"Can not open file: data.bin"<<endl;
             exit(1);
     }
     int n, x, y, low, mid, high;
     streampos p1, p2 = -1;
     infile.read((char*)&n, sizeof(int));      // 读入序列中的数值个数
     cout<<"Please input x: ";
     cin>>x;                                   // 输入待查找的x
     low=1; high=n;
     while(low<=high)
     {      mid=(low+high)/2;
            p1=mid*4;
            infile.seekg(p1);                  // 将读写位置指针移到中间数据的起始位置
            infile.read((char*)&y, sizeof(int));
            if(x==y){p2=p1; break;}            // 查找到数据
            else if(x<y) high=mid-1;
            else low=mid+1;
     }
     if(p2>0) cout<<"Position="<<p2/4<<endl;
     else cout<<"Not found!\n";
     infile.close( );
}
```

用二分法查找有序数列时效率较高，尤其当数据量比较大时更是如此。本程序在查找过程中，读取的数据个数较少。

## 14.8 输入／输出流的出错处理

在使用流进行输入／输出操作时，可能会出现一些错误。如当程序中需要输入整型数时，在输入流中却出现了字符；当已到达文件结尾处时，程序中还试图从文件中读取数据。C++提供了一套类机制，用于检测输入／输出流状态，以发现错误并清除错误状态。

在14.3.1节中给出的ios类的描述中，可以看到一个数据成员state，它是输入／输出流的出错状态字，它的每个二进制位表示一种输入／输出状态，各状态位由ios类中定义的枚举常量来表示：

```
enum io_state {  goodbit = 0x00,          // 表示状态正常
                 eofbit  = 0x01,          // 表示达到了文件结尾
                 failbit  = 0x02,         // 表示I/O操作失败
                 badbit  = 0x04           // 表示试图进行非法操作
              };
```

当状态是failbit时，表示输入／输出操作失败，如程序中要求输入整数，但在输入流中却出现了字符，无法正确完成输入操作，结果输入操作失败。当状态是badbit时，表示试图进行非法操作，如向只读文件中写数据。

ios类提供了几个内联函数，用于完成查看状态字state中的各个状态位、清除状态字等工作。它们是:

```
int  ios::rdstate( ) const { return state; }
int  ios::operator!( ) const { return state&(badbit|failbit); }
int  ios::bad( ) const { return state & badbit; }
void ios::clear(int _i=0){ lock( ); state = _i; unlock( ); }
int  ios::eof ( ) const { return state & eofbit; }
int  ios::fail( ) const { return state & (badbit | failbit); }
int  ios::good( ) const { return state == 0; }
```

从函数代码可以分析出，各函数的功能如下:

| | |
|---|---|
| rdstate( ) | 返回当前状态字 |
| operator!( ) | ! 运算符重载，与fail( )函数功能相同 |
| bad( ) | 如果badbit被置位，则返回真，否则返回假 |
| clear(int _i=0) | 用参数 _i 置新的状态字 |
| eof( ) | 如果eofbit被置位，则返回真，否则返回假 |
| fail( ) | 如果badbit 或failbit被置位，则返回真，否则返回假 |
| good( ) | 如果无任何错误则返回真，否则返回假 |

可以利用上述函数来检测流是否出错。

例如:
```
ifstream infile("in.dat");
if(infile.good( ))
{ cout<<"open success\n";  打开成功，在此行进行某些处理;}
```

例如:
```
ifstream infile("in2.dat",  ios::in|ios::nocreate);
if( infile.fail( ) )        // 或 if(! infile )，重载运算符!，与fail()
                            // 函数功能相同
{ cout<<"open fail\n";  打开不成功，在此行进行某些处理;}
```

例如:
```
while(! infile.eof( ) )
{ infile>>x ;  当未达到文件结尾时，在此处循环读取数据;}
```

例如:
```
infile.clear( );  清除文件流出错状态
cin.clear( );     清除标准输入流出错状态
```

例如，下面程序用于输入10个成绩并统计平均分数:

```
#include <iostream.h>
void main( )
{   double aver=0, score;
    for(int i=0; i<10; i++)
    {      cin>>score;
           aver+=score;
    }
    aver/=10;
    cout<<"aver="<<aver<<endl;
}
```

运行时，要求正确无误地输入10个数值，但实际操作时有可能输入错误，如若第3个数输入错误，则程序的一次执行过程可能如下:

```
80<Enter>
87<Enter>
a<Enter>   // 输入错误
aver=86.3
```

程序一旦遇到一个错误的输入，马上停止运行，结果不正确。

所以为了保证程序在遇到错误的输入时，能自动跳过错误，继续重新输入正确数据，在程序中必须加入一些出错检测及清除错误状态的操作。

**例14.31** 出错处理函数的使用。

```
#include <iostream.h>
void main( )
{   double aver=0, score;
    char temp[80];
    for(int i=0; i<10; i++)
    {       cin>>score;
            while(cin.rdstate( ))          // 若有输入错误
            {
                  cin.clear( );            // 清除出错状态，但错误输入依然留在输入缓冲区中
                  cin.getline(temp,80);    // 将缓冲区中的错误输入读入到temp，丢弃
                  cout<<temp<<"为非法输入，请重新输入!\n";
                  cin>>score;
            }
            aver+=score;
    }
    aver/=10;
    cout<<"aver="<<aver<<endl;
}
```

程序的一次运行状况可能如下:

```
10<Enter>
20<Enter>
30<Enter>
a<Enter>
a为非法输入，请重新输入!
40<Enter>
50<Enter>
60<Enter>
70<Enter>
80<Enter>
90<Enter>
100<Enter>
aver=55
```

## 习题14

1. 编制程序，创建文本文件mul.txt，其内容为九九乘法表，其格式为:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 |
| 3 | 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 |
| ..... | | | | | | | | | |
| 9 | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 |

要求通过循环自动生成该表，其中每个数据占4个字符位置，右对齐。

2. 用编辑器产生一个文本文件data.txt，其内容为若干实数，数据之间以空白字符分隔。编程从该文件中读入这些实数，求出这些实数的平均值，在程序中创建并产生一个文本文件result.dat，内容为data.txt中的全体实数，每行5个数，最后一行是求出的平均值。
3. 将0°~90°的每一角度的sin( )和cos( )值求出，写到二进制文件sincos.bin中。对每一角度，写出数据的格式为：以int型的二进制格式写出角度值，然后以double型的二进制格式连续写出对应的sin和cos值。提示：C++数学库函数sin( )和cos( )的参数是弧度。将“角度”转换成“弧度”的公式为：弧度 = (角度/180) × π。
4. 将上一题sincos.bin中的20°~30°之间每一度的度数及其sin和cos值读出，显示到屏幕上，要求一行写出一个度数的数据，各数据列对齐。
5. （1）重载插入和提取运算符：实现复数对象整体的输入和输出。（2）用构造函数实现：将一个double类型数据赋值给复数对象的类型转换。（3）重载类型转换函数：实现将一个复数类型对象赋值给double类型数据的转换。
6. 通过键盘输入一篇英文文章，然后将其存放到一个文本文件article.txt中。

# 第15章 模　板

在C++中可以利用模板技术来设计函数与类，使得这些用户定义的函数和类就好像是一块可变的模板一样，适应不同的情况。程序设计者利用模板功能便可以使得函数的返回值、接收的参数或者类的数据成员成为程序设计者所希望的任何一种数据类型。模板功能不但大幅度提高了函数与类的灵活性，同时也减轻了程序设计者维护程序的工作量，从此程序员可以利用模板技术来结束对大量类似代码的重复书写工作。本章主要介绍模板的定义和使用方法。

## 15.1 概述

什么是模板（template）？其重要性如何?

在程序应用中，经常会遇到同样的一段代码需要用来处理不同类型的数据的情况。传统的处理方法是用文本编辑器（各种编程环境中提供的源代码编辑器）复制程序代码，并且为其加上不同的函数名称，将其间涉及数据类型的部分用手工进行修改。这样做，会造成同样功能的代码，仅仅由于需要处理不同的数据类型，就重复书写多次的情况。而且当这些内容雷同的代码的功能需要改变时，程序员又必须要对所有重复书写的相关代码段进行重复的维护。可以看出这种原始的复制方法在编写和维护程序时效率十分低下，并且可靠性较低。通过前面章节对C++的学习，我们还可以使用重载函数技术，将代码段的功能封装在函数中，那么就可以仍旧使用相同的函数名称来处理不同数据类型的数据。但是必须清醒地认识到函数重载的本质，函数重载依然是一种人工复制代码段的方法，它的使用只能使得我们在定义函数时不再另外给函数起名，在调用函数时书写简洁，使我们不再需要用不同的函数名来对应同样功能的函数而已。函数的重载并没有降低我们书写程序和维护程序的工作量，当需要修改函数功能时，必须对所有同名的重载函数中完全相同的算法进行完全相同的维护。在多处出现复制代码的情况下（包括重载函数），经常会出现由于程序员的疏忽而造成修改遗漏的情况，这样会为程序的运行带来极大的隐患。例如，我们要实现对于不同的数据类型求两个数的最大值的算法，一般的实现方法如例15.1所示。

**例15.1** 演示函数重载判断多种类型数值的大小。

```
#include  <iostream.h>
int  max(int num1, int num2)
{
   return  num1 > num2 ? num1 : num2;
}
double  max(double d1, double d2)
{
   return  d1 > d2 ? d1 : d2;
}
char*  max(char* str1, char* str2)
{
   return  *str1 > *str2 ? str1 : str2;          // 仅仅比较两个字符串的第一个字符
}
void  main( )
{
    int     m1 = max(100, 200);                  // 调用max(int, int)
    double  m2 = max(7.8, 5.1);                  // 调用max(double, double)
    char*   m3 = max("Kate", "Mary");            // 调用max(char*, char*)
```

```
    cout << "The maxium of 100 and 200 is: " << m1 << endl;
    cout << "The maxium of 7.8 and 5.1 is: " << m2 << endl;
    cout << "The maxium of 'Mary' and 'Kate' is: " << m3 << endl;
}
```

程序例15.1定义了三个重载函数，其差别只在于参数的数据类型与返回值的数据类型不一样。这样函数max便可以适用于整数、浮点数与字符串，当程序中有如下语句时：

```
double  m2 = max(7.8, 5.1);
```

编译器便会调用下列重载函数，以求得double类型参数的最大值：

```
double  max(double d1, double d2);
```

由程序例15.1获得的输出结果如下：

```
The maxium of 100 and 200 is:  200
The maxium of 7.8 and 5.1 is:   7.8
The maxium of 'Mary' and 'Kate' is:  Mary
```

我们在第6章的编译预处理中介绍过C++语言对上述问题的另一种处理办法：使用宏。使用宏，虽然你可以对于相同的算法只写一次程序代码，系统会在编译预处理中自动对源代码进行替换，防止人工疏漏造成的错误，但是宏也存在它自身不可克服的缺点：

（1）宏只能实现简单的功能。

（2）宏只做机械式的替换，不提供数据类型的检验，因此使得C++编译器中先进的数据类型检测机制被避开了。由此丧失了C++提供的一个十分重要的功能。这对于C++程序来说，是一个非常严重的缺陷。例如：

```
#define  max(x,y)  ( ((x) > (y))  ? (x) : (y) )
```

原意是对两个兼容的数据类型进行比较，但是使用宏定义以后，使得int和struct这两种不兼容的数据类型也可以进行比较。这样的程序是无意义的。

（3）宏定义并非函数，程序中任何调用宏的地方都会被编译器预处理机制插入宏体所定义的那一段代码，而非只是一个函数调用。因此你在源代码中每使用一次宏，编译预处理以后的源代码的尺寸就会膨胀一点，这样会造成最终执行程序的尺寸较大。

对于上面讨论的一系列问题，C++为我们提供了一个更好的、由系统来实现的复制方法：模板功能。可以使用这个技术来解决人工复制代码段和使用宏定义所引起的种种问题。

模板技术把功能代码段的“算法”和算法中的“计算数据的数据类型”区分开来。你可以先写算法的程序代码，而在定义算法时不确定实际计算数据的数据类型，然后在实际使用算法时再根据实际获得的数据类型来确定使用何种数据类型进行数值计算。

C++模板语法使得“数据类型”也可以以参数的形式出现，在实际使用时，可以将某种数据类型作为参数传递给模板。有了模板，你可以拥有宏的“只写一次”，系统自动替换的优点；也可以获得重载函数的“数据类型检验”的优点。

C++的模板有两种，一种是针对函数（function）的，另一种是针对类（class）的，下面我们分别对其进行介绍。

在正式学习模板技术之前，我们可以将其与前面章节所介绍的面向对象技术做一个类比：类是对象的抽象与描述，而函数模板是函数的抽象，类模板是类的抽象。按照这个思路，我们可以比较容易理解和掌握后续章节的内容。

## 15.2　函数模板

假设我们需要写一个求两个数中的大数的函数，名曰max。简单定义如下：

```
int  max(int num1, int num2);
```

可以看出，此函数只适用于求两个整数的最大值，对于C++的其他数据类型则不适用，例如：

```
int  maxium = max(10, 20);            // 正确
double  maxium = max(4.21, 4.99);     // 有问题
```

当我们给函数max两个非整数类型的参数时，编译器会自动按照第2章所描述的转换规则进行数据类型的转换，若不能转换则报错。即使可以转换也不代表程序可以得到正确的结果，因为4.21和4.99按照规则经过数据类型转换后，都舍弃了小数部分，变成了整数4，无法辨别谁的值为最大值。因此，要想获得正确的结果，一般的方法是使用函数的重载功能。具体实现如例15.1的程序代码所示。

从例15.1可以发现，使用重载技术的确可以在不同数据类型的数据进行比较时取得正确的结果。但是，这时为了符合不同数据类型的参数要编写多个算法完全一样的重载函数，这对于程序设计和维护都是一种不必要的负担。其实，在前面的例子中，每一个重载函数的实际操作定义部分，除了所使用的参数类型以外，可以说是完全一样的。如果能够把数据类型和算法代码相分离，将数据类型也变成参数之一，在使用时临时指定数据类型，就可以实现算法代码与多种数据类型之间的多个对应关系。同一段算法代码就可以“以不变应万变”地处理多种数据类型了。

如本章开始所述，C++中提供了关键字template的功能，正是按照上面的思路来解决这种相同操作代码处理不同数据类型的问题，这项技术就称为函数模板（Function Templates）。函数模板的特定实例叫做模板函数（Template Functions）。

### 15.2.1 定义函数模板

利用关键字template按照特定的格式来书写函数，可以在函数中将数据类型也作为参数，这种特殊的函数就称为函数模板。函数模板能够依据在程序运行过程中被使用的情况，来动态地决定这些数据类型参数当时真正的数据类型，而其返回值的数据类型也同样可以利用模板的方式来动态地确定。定义函数模板的格式如下：

```
template <class Type>
<Type> <FunctionName> (<Parameter List>)
{
    //函数体
}
```

上述的定义是一个整体，不过为了方便理解和记忆，我们可以将其分为两个部分来观察和学习，具体如下：

```
template < class Type >                          // 模板定义部分
```

与

```
<Type> <FunctionName> (<Parameter List>)         // 函数定义部分
{
    //函数体
}
```

函数模板的定义以关键字template开始，接着在符号‘<’和‘>’之间定义该函数的数据类型参数。由于在定义时函数的真正数据类型并未确定，必须在以后根据该函数在程序运行中被如何使用而定，因此被定义的数据类型参数Type可以说是模板的形式参数，其名称由设计者自己决定（本例中其名称定义为Type）。此参数的前面必须加上关键字class，以表示该参数可以是C++的任何一个基本数据类型，还可以是用户自己定义的类或者是其他数据类型。注意这里使用的关键字class并不一定表示C++的类，它也可以是一个基本的数据类型，如int、float、double等。<class Type>只不过是表示：Type是一种数据类型，而此数据类型将在实际调用此函数时才确定。由于<class Type>为关键字template的参数列表，在模板中至少要定义一个数据类型参数，因此不写参数列表或者参数列表中没有定义任何数据类型参数都会出错。

在模板定义部分定义了一个或者多个数据类型参数以后，紧接着就要定义一个特殊的函数来使用这些数据类型参数，我们认为这后面的部分是函数模板的函数定义部分。这部分的定义与一般函数的定义

相同，首先是函数的返回值类型，书写成<Type>是指使用数据类型参数来动态地指定函数的返回值类型。接着是函数的名称<FunctionName>和函数的形参列表(<Parameter List>)。在形参列表中形参的类型可以是数据类型参数，也可以是普通的数据类型。例如:

```
template <class Type> <Type> <FunctionName> (Type a, Type b, int c)
{
    // 函数体
}
```

当编译系统发现有一个函数调用如下:

```
<函数名>(<实参表>);
```

而且函数名与某个函数模板中定义的函数名相同时，编译器会将实参表中的数据类型传递给函数模板，根据当时的数据类型由函数模板生成一个重载函数，即模板函数。该模板函数的定义体与函数模板的函数定义体相同，而函数模板中的<class Type>（数据类型参数表）中的数据类型参数的确定是以函数调用处的<实参表>的实际数据类型为依据的。“模板函数”是“函数模板”的实例化，是由系统自动管理的。

与一般函数的形式参数一样，在实际调用函数时确定了数据类型参数Type的具体类型以后，可以将其作为函数<FunctionName>的真正的数据类型来使用。确定数据类型的过程其实就是一个数据类型转换的过程，对于函数的定义部分:

```
<Type> <FunctionName> (Type a, Type b, ...)
```

当Type为int时，函数会被转换为:

```
int  <FunctionName> (int a, int b, ... )
```

当Type为double时，函数会被转换为:

```
double <FunctionName> (double a, double b, ... )
```

下面在程序例15.2中，用一个实际的函数模板应用程序为例，对函数模板的使用进行更加详细的说明。

**例15.2**　演示使用模板函数判断多种类型数值的大数。

```
#include  <iostream.h>
template  <class Type>                    // 定义函数模板的模板说明部分
Type  max(Type var1, Type var2)           // 定义函数模板的函数说明部分
{
    return  var1 > var2 ? var1 : var2;
}
void  main( )
{
    int     m1 = max(100, 200);           // 函数模板生成max(int, int) 模板函数
    double  m2 = max(7.8, 5.1);           // 函数模板生成max(double, double) 模板函数
    char*   m3 = max("Kate", "Mary");     // 函数模板生成max(char*, char*) 模板函数
    cout << "The maxium of 100 and 200 is:  " << m1 << endl;
    cout << "The maxium of 7.8 and 5.1 is:  " << m2 << endl;
    cout << "The maxium of 'Mary' and 'Kate' is:  " << m3 << endl;
}
```

程序例15.2与例15.1实现的功能基本上是一样的。唯一的差别在于程序例15.1中用重载技术实现的max( )函数在程序例15.2中被改写成了一个函数模板，用来实现对各种不同的数据类型的数据做完全相同的处理。

模板函数使用的是数据类型代换的技术，它可以通过一个函数模板来动态生成一系列具有相同算法而数据类型不同的函数。它的实现原理虽然与函数重载技术不同，但它们都是通过利用计算机高速运算的能力来管理多个函数的技术，它们都能够提高程序编写和维护的效率，只不过应用的范围不同而已。

函数重载技术可以使得被重载的各个函数的函数体实现不同的算法，而模板函数技术则不可以如此，因此在实现同名函数处理的时候，要注意根据具体情况选择不同的实现技术。

函数max( )被设计成函数模板，其两个参数和返回值的数据类型都为数据类型参数Type。此Type数据类型参数所代表的真正数据类型只有到该函数被真正调用的时候才能确定。在例15.2中以下语句:

```
int  m1 = max(100, 200);
```

所给予的两个参数的类型都是int类型，因此编译器将Type视为int，用int替换原来template中的Type，替换以后的结果如下:

```
int  max(int var1, int var2)          // 系统自动生成的模板函数
{
   return  var1 > var2 ? var1 : var2;
}
```

同理，语句:

```
char*   m3 = max("Kate","Mary");
```

也会被编译器自动替换为:

```
char*  max(char* var1, char* var2)    // 系统自动生成的模板函数
{
  return  var1 > var2 ? var1 : var2;
  //此处操作与例15.1略有不同，将在15.2.2节中说明。
}
```

注意:

(1) Type的实际数据类型是由编译器根据调用函数时传递给函数的实际参数的数据类型来决定的，而不是根据函数的返回值的数据类型来决定的。如以下语句:

```
int  x = max(7.8, 5.1);
```

由于两个实际参数都是浮点数类型，编译器将Type转换为浮点数类型double，返回值的类型也同样是浮点数类型。若将返回值赋给整数变量x，则编译器会发出警告，提醒程序设计者返回值的小数部分会丢失。

(2) 由于定义函数模板max时，所设置的两个参数的类型是一样的。因此，当调用max生成模板函数时，若给予模板的两个实际参数的数据类型不一样，将发生错误。例如:

```
int  val = max(7.8, 30);              // 错误
```

此时，编译器会先将Type解释成double类型，但是当它发现第二次出现Type的地方不能再将其解释成double类型时，便发生错误。当模板函数中的某个数据类型参数一旦被确认后，便不可以再改变。这时，可以使用第2章中所介绍的强制类型转换运算符来改变不符合已经被确认的数据类型的变量。例如，以下语句就可以避免上述错误:

```
double  val = max(7.8, (double)30);
```

我们在前面提到过，使用宏也能做到系统自动替换的功能。但是宏只做机械式的替换，不提供数据类型的检验，因此使得C++编译器中先进的数据类型检测机制被避开了。而通过上面的例子可以看出，当使用模板技术时，C++编译器使用了数据类型检测机制，可以预先防止程序的书写错误。

在定义和使用函数模板时，特别容易造成指针的使用错误。下面通过程序例15.3来说明在函数模板中指针的正确使用方法。

**例15.3** 演示函数模板中指针的用法。

```
#include  <iostream.h>
template  <class Type>
Type  Max(Type* array, int size=0)
```

```
{
    Type max = array[0];
    for(int i=1; i<size; i++)
       if(max < array[i])
          max = array[i];
    return  max;
}
void  main( )
{
    int     Array1[ ] = {1, 0, 32, 4, 5, 16, 7, 28, 9, 12};
    double  Array2[ ] = {1.2, 22.3, 3.4, 0.4, 5.6, 6.7, 17.8, 8.9, 7.9, 10.1};
    int     max1 = Max(Array1, 10);    // 函数模板生成Max(int* array, int size) 模板函数
    double  max2 = Max(Array2, 10);    // 函数模板生成Max(double* array, int size) 模板函数
    cout << "The maximum number in integer array is: " << max1 << endl;
    cout << "The maximum number in double floating array is: " << max2 << endl;
}
```

程序例15.3中定义了一个用来求一个数组中所有元素的最大值的函数Max( )，由于数值数组可能是整数数组或者浮点数数组等，而求数组中所有元素的最大值的算法是一样的，因此采用函数模板技术来替代重载函数等方法是一种比较好的选择。

函数模板Max中定义了一个数据类型为Type的变量array和一个整型变量size，其中array是一个指向数据类型参数Type的指针变量。必须注意，array是Max( )函数的形式参数并且是指针类型，因此当调用函数Max( )的时候，第一个参数必须是指针变量或者数组名。例如:

```
int  Array1[ ] = {1, 0, 32, 4, 5, 16, 7, 28, 9, 12};
max1 = Max(Array1, 10);
```

函数模板除了定义的方式不同于一般函数外，其他的书写规则仍与一般函数相同。它的函数定义部分也可以被定义为static、extern、inline函数。例如:

```
template  <class Type>  inline  Type  max(Type var1, Type var2)  {...}
template  <class Type>  static  Type  Sum(Type* array, int size=0)  {...}
```

注意: 关键字inline、static等必须在template <class Type>之后，以下的语句写法是错误的:

```
inline  template  <class Type>  Type  max(Type var1, Type var2)  {...}
static  template  <class Type>  Type  Sum(Type* array, int size=0)  {...}
```

### 15.2.2 模板函数的重载

如果用户运行前面的程序例15.2，会发现输出的结果与程序例15.1完全一样，函数模板的确能够解决处理不同数据类型的问题，并且整个程序的设计和维护效率也大幅度提高了。但是如果在程序例15.2中将语句:

```
char*   m3 = max("Kate","Mary");
```

改写为:

```
char*   m3 = max("Mary","Kate");
```

则输出结果变为:

```
The maxium of 'Mary' and 'Kate' is:  Kate
```

根据程序例15.1的要求，我们要求函数max可以对两个字符串的第一个字母进行比较，返回第一个字母的ASCII码较大的字符串。所以正确的结果应该是字符串“Mary”。而上面的程序结果却非我们所预期的结果，这说明指针变量m3的内容并非正确结果“Mary”。

为什么会发生这种情况呢？仔细观察程序例15.2中的函数模板max的定义便可以发现出现错误的原因:

```
template  <class Type>  Type  max(Type var1, Type var2)
{
    return  var1 > var2 ? var1 : var2;
}
```

当max函数的实际参数为字符串变量时，类型参数Type被编译器替换成char*，生成的模板函数max( )被解释成:

```
char*  max(char* var1, char* var2)
{
    return  var1 > var2 ? var1 : var2;
}
```

根据第9章介绍的指针定义，此时形式参数var1和var2的内容分别为两字符串的起始地址，而非字符串的第一个字母。程序执行的功能也就变成了将参数var1和var2中存放的地址进行比较，而非字符串的第一个字母的比较。并且由于参数var2定义在var1的后面，它所对应的存储器中的地址要比var1所对应的存储器的地址高，相对的数值较大。因此，将参数var1和var2中存放的地址进行比较时，永远都是var2大于var1，在程序例15.2中返回的字符串也永远都是两个字符串中的后者。

当函数模板无法对于每一种数据类型都得到预期的结果时，可以特别为某种数据类型设计一个与该函数模板同名称的一般函数，而此一般函数的形式参数列表与函数模板必须是一致的。该一般函数与函数模板的差别在于此一般函数的所有形式参数与返回值的数据类型都已经明确地确定了，而且函数体中的算法流程也可以与函数模板中定义的不同。它相当于在函数模板生成的一系列模板函数之外又定义了一个特殊的重载函数，因此我们将其称为重载模板函数。例如，对程序例15.2的改进便可以采用为字符指针参数提供一个进行特别处理的重载模板函数。程序例15.4实现了这种改进。

**例15.4** 演示重载模板函数的实现。

```
#include  <iostream.h>
template  <class Type>  Type  max(Type var1, Type var2)  // 函数模板
{
    return  var1 > var2 ? var1 : var2;
}
// 模板函数的重载
char*  max(char* str1, char* str2)
{
    return  *str1 > *str2 ? str1 : str2;
}
void  main( )
{
    int     m1 = max(100, 200);          // 函数模板生成max(int, int) 模板函数
    double  m2 = max(7.8, 5.1);          // 函数模板生成max(double, double) 模板函数
    char*   m3 = max("Mary","Kate");     // 重载模板函数max(char*, char*)
    cout << "The maximum of 100 and 200 is:  "
         << m1 << endl;
    cout << "The maximum of 7.8 and 5.1 is:  "
         << m2 << endl;
    cout << "The maximum of 'Mary' and 'Kate' is:  "
         << m3 << endl;
}
```

在程序例15.4中除了原来程序例15.2中的函数模板max外，为了特别处理字符串比较的情况，又编写了一个重载模板函数:

```
char*  max(char* str1, char* str2)
{
    return  *str1 > *str2 ? str1 : str2;
}
```

由于是将一般函数与模板函数进行重载，当调用max( )函数的时候，系统会优先匹配非模板函数的函数max( )，如果在非模板函数的函数中找不到匹配的函数，系统才会对函数模板进行参数转换匹配，产生对应的模板函数。可以认为在C++中，被重载的同名函数中的一般函数的优先级要高于模板函数。因此，在程序例15.4中系统会针对语句:

```
char*   m3 = max("Mary","Kate");
```

调用我们新增加的特别用来处理字符串数据的重载模板函数，而对其他语句则由于数据类型非字符指针类型，无法与这个唯一的重载模板函数相匹配，只能调用由函数模板转换获得的模板函数。这时再运行程序，用户可以发现无论字符串的位置和内容如何变化，程序都能够得到正确的结果了。

同样道理，我们也可以使用重载模板函数的方法来实现字符串完全比较的功能。只需要增加重载函数如下即可（注意：由于使用了字符串比较系统函数，因此需要包含"string.h"头文件):

```
char*  max(char* str1, char* str2)
{
    return  strcmp(str1,str2) > 0 ? str1 : str2;
}
```

注意：编写重载模板函数来解决函数模板无法对某一个特别的数据类型进行统一处理的计算时，必须要保证这个重载函数与原来的模板函数有相同的函数名称。而且返回值的数据类型以及形式参数列表中的数据类型的排列顺序的对应关系，也必须与原来的模板函数一致。即编写的重载模板函数应当被看做是函数模板的一个特例。

### 15.2.3　函数模板的重载

在15.2.2节中介绍的函数模板的特殊处理方法，其实就是函数模板与一般函数的重载。除此以外，函数模板之间也可以进行重载。本节就对这一技术进行简单的介绍。

函数模板的重载与一般函数的重载在概念和方法上是完全一样的，利用函数模板的形参列表的不同，可以为函数模板提供重载。例如以下函数模板multiply的定义便说明了如何实现函数模板的重载技术:

```
template   <class Type>   Type  multiply(Type var1, Type var2)
template   <class Type>   Type  multiply(Type var1, int var2)
template   <class Type>   Type  multiply(Type var1, char* var2)
```

可以看到，上面定义的三个函数模板，函数名称、返回值的类型完全相同，只是形参列表中的参数类型不完全相同。这和我们前面学过的普通函数的重载语法是完全一致的。

利用函数模板的重载不仅可以解决不同数据类型的参数问题，还可以解决参数个数不同的问题，使得函数的应用能力进一步地增强。

必须注意，一个函数是否可以被重载，在于函数的形参列表是否相同而不在于返回值是否相同。同理，以下两个函数模板就并非是重载的关系:

```
template  <class Type>  Type  multiply(Type var1, Type var2)
template  <class Type>  Type*  multiply(Type var1, Type var2)
```

在设计函数模板时容易犯的另外一个错误是，认为以下两个函数模板是重载的关系:

```
template  <class T1>  T1  multiply(T1 var1, T1 var2)
template  <class T2>  T2  multiply(T2 var1, T2 var2)
```

前面说过，数据类型参数T1和T2所代表的实际数据类型只是在函数被真正调用时才由编译器确定，数据类型参数的名称本身没有任何实际意义，即参数名是T1或者T2并没有任何差别。因此上面的两个函

数模板其实是一模一样的函数模板，如果将上面的两个定义写在同一个程序段中，就会造成函数模板的重复定义，编译器一般会在编译的过程中发现这个错误并向用户报警。程序例15.5说明了如何在应用程序中实现函数模板的重载。

例15.5 演示重载函数模板的实现。

```
#include  <iostream.h>
template  <class Type>  Type  Max(Type* array, int size=0)
{
   Type m = array[0];
   for(int i=1; i<size; i++)
      if(m < array[i])
         m = array[i];
   return  m;
}
// 重载函数模板
template  <class Type>  Type  Max(Type* array1, Type* array2, int size=0)
{
   Type m = array1[0];
   if(m < array2[0])
      m = array2[0];
   for(int i=1; i<size; i++)
   {
        if(m < array1[i])  m = array1[i];
        if(m < array2[i])  m = array2[i];
   }
   return  m;
}
int  Array1[ ] = {1, 0, 32, 4, 5, 16, 7, 28, 9, 12};
int  Array2[ ] = {11, 120, 31, 14, 52, 16, 77, 18, -9, 20};
double  Array3[ ] = {1.2, 22.3, 3.4, 0.4, 5.6, 6.7, 17.8, 8.9, 7.9, 10.1};
void  main( )
{
    int     max1 = Max(Array1, 10);
    double  max2 = Max(Array3, 10);
    int     max3 = Max(Array1, Array2, 10);
    cout << "The maximum number in integer array1 is: " << max1 << endl;
    cout << "The maximum number in double floating array is: " << max2 << endl;
    cout << "The maximum number in two integer arrays is: " << max3 << endl;
    getch( );
}
```

程序例15.5是在程序例15.3的基础上再加入一个重载的Max函数模板修改而成的。原来的Max函数模板是用来求一个数组中所有元素的最大值，新加入的Max函数模板扩展了形式参数的定义，可以接收两个数据类型相同的数组，并且求两个数组中的所有元素的最大值。与原来的Max函数模板一样，它也要指明每个需要求和的数组元素的个数。

### 15.2.4 多重数据类型参数的模板函数

在前面所有的例子中，函数模板都仅仅含有一个数据类型的参数。而在C++的函数模板技术中，其实是可以允许一个函数模板同时具有多个数据类型参数的。要使得函数模板同时具有多个数据类型参数，以便有更强大的处理能力去处理不同数据类型的情况，函数模板的格式定义如下:

```
template  <class Type1, class Type2>
Type1  func(Type1 a, Type2 b)
{
   // 函数体
}
```

这个定义同样可以理解为由两部分组成（注意：其实我们分开理解的两部分是一个整体），具体如下所示：

```
template  <class Type1, class Type2>                    // 模板的定义部分
```

和

```
Type1  func(Type1 a, Type2 b)                           // 函数的定义部分
{
    //函数体
}
```

可以看出，多重数据类型参数的函数模板的定义与前面所讨论的单个数据类型参数的函数模板的定义非常类似，它们都是由两部分组成。而不同之处在于，多重数据类型参数的函数模板在定义中增加了一个数据类型参数Type2。Type1和Type2是可以分别代表不同数据类型的类型参数。由于函数的返回值只能是唯一的，因此它的数据类型只可以是Type1或者Type2中的某一个（在前面的定义中，函数的返回值的类型就被定义为Type1）。此外，形式参数和返回值的数据类型还可以是设计者直接指明的如int、double等非Type1和Type2类型的其他数据类型。

在定义多重数据类型参数时，必须注意以下几点：

（1）在定义数据类型参数时，每一个数据类型参数的前面都必须书写一个关键字class，以下的定义格式是错误的：

```
template  <class Type1, Type2>  Type1  func(Type1 a, Type2 b)  {...}
```

在数据类型参数Type2的前面遗漏了关键字class，这样的程序是无法编译通过的。

（2）类型参数Type1与Type2虽然名称不同并且可以代表不同的数据类型，但是它们也可以是相同的数据类型（根据调用时的实参类型，它们可以被替换为相同的数据类型）。具体应用如程序例15.6所示。

**例15.6**　演示多重数据类型参数的函数模板的使用。

```
#include  <iostream.h>
template  < class Type1, class Type2>  Type1  max(Type1 var1, Type2 var2)
{
    return  var1 > var2 ? var1 : var2;
}
void  main( )
{
    int  m1 = max(30,7.56);       // 函数模板被解释成max(int, double) 模板函数
    int  m2 = max(70, 'k');       // 函数模板被解释成max(int, char) 模板函数
    int  m3 = max(100,300);       // 函数模板被解释成max(int, int) 模板函数
    cout << "The maxium of 30 and 7.56 is: " << m1 << endl;
    cout << "The maxium of 70 and 'k' is: " << m2 << endl;
    cout << "The maxium of 100 and 300 is: " << m3 << endl;
}
```

在程序例15.6中，前两次max( )函数的调用都将不同数据类型的参数传递给了函数。在函数调用时，数据类型参数一旦被匹配转换后，便不可以再被改变。前两次max函数中的Type1和Type2都只被转换一次，它们都可以拥有自己的数据类型，数据类型互不相同。但是C++并没有规定后进行转换的Type2的数据类型必须与先前转换的Type1的数据类型不同，因此，Type1和Type2转换后的数据类型也可以是相同的。

如第三次max函数的调用:

```
int  m3 = max(100,300);
```

此时，调用函数max( )时，Type1和Type2就都被解释成int类型。由于多重数据类型参数的函数模板的这个特性，往往会在使用的时候造成错误。如下列语句所示:

```
template  <class Type1, class Type2>  Type1  Max(Type1 var1, Type2 var2)
{
    return  var1 > var2 ? var1 : var2;
}
template  <class Type>  Type  Max(Type var1, Type var2)
{
    return  var1 > var2 ? var1 : var2;
}
```

初看起来，上面的两个函数模板并不相同，完全可以并存。但是下面的语句却会造成编译器在调用函数时出现函数不确定的情况:

```
int  m1 = max(10,20);
```

因为调用函数时两个函数模板都符合被调用的条件，所以编译器不能确定到底应该使用哪个函数模板。因此用户在使用多重数据类型参数的函数模板时必须要特别注意这种错误。解决这个问题的方法十分简单。由于具有多重数据类型参数的函数模板既可以符合参数类型相同的情况，又可以符合参数类型不相同的情况，较之参数数据类型必须相同的单一数据类型参数的函数模板有许多灵活性；因此，当程序中同时存在同名的多重数据类型参数的函数模板和单一数据类型参数的函数模板时，只要保留多重数据类型参数的函数模板即可。

## 15.3 类模板

前面小节介绍的函数模板（Function Templates）主要是设计用于解决函数的参数为各种不同的数据类型，而函数体的算法又基本相同时程序的高效率实现问题，类模板（Class Templates）也是为了同样的目的而设计的。虽然类模板和函数模板的设计目的一致，但是从字面上也可以看出类模板与函数模板的应用目的是不同的，所以它们之间在实现上有着许多的不同，设计类模板要比设计函数模板复杂和困难得多。例如，函数模板的调用是由编译器根据用户传递的实际参数自动选择匹配并且进行参数替换的，而类模板在使用的时候则必须由程序设计者自己指明该类模板希望按照哪一种数据类型去解释生成一个实际的类（模板类），这是在使用上的极大不同。除此之外，其他在诸如定义的格式等方面函数模板和类模板之间也有许多不同。

### 15.3.1 定义类模板

假设有如下类:

```
class  iVector
{
private:
  int*  data;
  int   size;
public:
  iVector(int size)  {...}
...
}
```

用户可以利用这个类去表示和操作整数数组。由于数组元素的数据类型确定，因此程序设计者可以很方便地根据元素的个数进行内存的分配和回收处理（因为我们知道每个整数类型的数据在16位操作系

统中占用两个字节的内存空间，在32位操作系统中占用四个字节的内存空间）。数据元素的类型确定方便了类的设计和使用，但是同时也带来了一个弊端：这个类只能适用于整数数组。对于C++中其他数据类型的数组则必须重新定义一个新的类来表示。例如double类型的数组类定义如下：

```
class  dVector
{
private:
  double*  data;
  int      size;
public:
  dVector(int size)  {...}
...
}
```

与本章开始所描述的函数的重载一样，对于每一个新增加的数据类型不同的类，程序设计者所要做的工作仅仅是将原来的类的程序代码再复制一份，然后修改其中与数据类型相关的部分代码，如声明数据成员的部分、成员函数的参数以及类的名称。可以想象，这样的复制对程序的编写效率以及以后程序的维护都是很不利的，一旦程序的操作方法需要变动，程序设计者就不得不将所有由复制生成的类的相同的代码都修改一遍，这其中还要保证不能够有任何的遗漏和错误。

如同利用函数模板技术可以解决如何用一个函数定义处理不同数据类型的数据的问题一样，利用类模板也可以解决如何让一个类定义能够处理不同数据类型的数据的问题。通过一个类模板，可以生成一系列数据类型不同的模板类，类模板与模板类的关系如同函数模板与模板函数的关系，都是抽象与具体的关系。

类模板的定义格式如下：

```
template  <class Type>  class  <ClassName>
{
    //类成员定义
};
```

类模板的定义与函数模板的定义类似，以关键字template开始，接着是在符号‘<’和‘>’之间定义数据类型参数列表，最后是与一般类相同的类体的定义部分。数据类型参数Type经过定义后，便可以当做实际数据类型用来定义类体中的成员函数和数据成员。如以下代码段所示：

```
template  <class Type>  class  Vector
{
private:
   Type*  data;                          // 数据成员
   int    size;
public:
   Vector(int n);
   Type  GetMem (Type a, Type b);        // 成员函数
   ...
}
```

其中，不包含数据类型参数的成员函数的定义格式如下所示：

```
template  <class Type>  Vector <Type>  ::  Vector(int n)
{
    //函数体
}
```

而包含数据类型参数的成员函数的定义格式如下所示：

```
template  <class Type>  Type  Vector  <Type>  ::  GetMem (Type a, Type b)
{
// 函数体
}
```

这些成员函数的定义格式看起来比一般类中成员函数的定义格式要复杂得多，其具体含义我们将在后面结合具体的程序实例来进行介绍。

可以看到，在上面的类模板的定义中使用了两次关键字class。需要注意的是，这里出现的两个关键字class具有不同的含义。数据类型参数列表中的class（即在<class Type>中）表示Type是一种数据类型，而类名Vector前面的class则是用来表示Vector是一个C++的类。读者在书写和阅读时要防止由于混淆含义而造成的错误。

在完成类模板的定义后，只要以下列方式定义类的对象，便可以自动产生具有不同数据类型的模板类，同时生成该类的对象。

```
Vector <int>  iVector(100);              // 定义整数类型对象
Vector <double>  dVector(100);           // 定义双精度浮点数类型对象
```

由上面的使用可以看出，类模板在使用的时候必须由程序设计者自己指明该类模板希望被解释成哪一种数据类型的模板类。而在前面介绍的函数模板的使用中，其数据类型的确定是由编译器根据用户传递的实际参数的数据类型自动进行替换的。这是类模板与函数模板之间的主要差别之一。

除了写法不同之外，类模板与函数模板的实现原理还是一样的。它们都是利用了数据类型的动态替换技术，用一个类模板或者函数模板在程序运行的时候动态地生成一系列数据类型不同的对象或者函数。它们生成的多个对象或者函数之间，彼此独立，拥有自己的运行空间。所以，在说明模板时必须加入足够的信息，以防止计算机在具体生成对象或者函数时发生错误。这也导致了模板定义书写格式的复杂性。在下面的程序例15.7中，我们将完整、具体地说明类模板的定义和使用。

**例15.7** 演示类模板的使用。

```
#include  <iostream.h>
#include  <iomanip.h>

template  < class Type >  class  Vector
{
private:
   Type*  data;
   int    size;
public:
   Vector(const int s);
   ~Vector( );
   Type&  operator [ ] (const int index);
   void  operator = (Type value);
};

template  <class Type>  inline  Vector <Type>  ::  Vector(const int s)
{
   size = s;
   data = new Type[size];
   for(int i=0; i<size; i++)
       data[i] = 0;
}
template  <class Type>  inline  Vector  <Type>  ::  ~Vector( )
{
```

```
    delete  data;
}
template  <class Type>  Type&  Vector  <Type> :: opérator[ ](const int index)
{
    if(index>=size || index<0)
    { cout<< "数组下标越界错误! " << endl;
      return(0);
    }
    else
      return  data[index];
}
template  <class Type>  void  Vector  <Type>  ::  operator = (Type value)
{
    for(int i=0; i<size; i++)
      data[i] = value;
}
void  main( )
{
     int num;
     cout << "请输入需要建立的数组的元素个数: ";
     cin >> num;
     // 生成类的对象
     Vector <int>     iVector(num);              // 整数类型对象
     Vector <double>  dVector(num);              // 双精度浮点数类型对象
     Vector <char>    cVector(num);              // 字符类型对象
     for(int i=0; i<num; i++)
     {
         iVector[i] = i;
         dVector[i] = i;
     }
     cVector ='A';
     cout.setf(ios::showpoint);                  // 显示浮点数的小数部分
     cout << setw(10) << "iVector" << setw(15) << "dVector" << "    " << "cVector" << endl;
     for(i=0; i<num; i++)
     {
         cout << setw(10) << iVector[i] << setw(15) << dVector[i]<< "    " << cVector[i] << endl;
     }
}
```

在程序例15.7中定义了一个简单的类模板Vector，其中数据成员data是一个用类的数据类型参数定义的指针变量，用以指向数组的起始地址。指针变量data的数据类型为模板的数据类型参数列表中所定义的数据类型参数Type。在类模板中还使用了运算符重载技术定义了两个运算符成员函数，其返回值分别为数据类型参数的引用和void类型:

```
Type&  operator [ ] (const int index);
void  operator = (Type value);
```

在定义了类模板的具体结构以后，还需要分别定义类的成员函数的函数体。下面以类模板的构造函数为例，说明如何在类模板的定义之外来定义类的成员函数:

```
template  <class Type>  inline  Vector  <Type>  ::  Vector(const int s)
{
   size = s;
```

```
    data = new Type[size];
    for(int i=0; i<size; i++)
    data[i] = 0;
}
```

定义类模板的成员函数首先与定义一般的函数模板一样，必须以关键字template开始，然后是用一对"<>"符号括起来的数据类型参数序列。特别要注意的是：这部分的定义必须要与成员函数所属的类模板的数据类型参数定义部分完全一致。例如：

```
template <class Type> class Vector {...};          // 模板类
template <class Type> inline
Vector <Type> :: Vector(const int s) {...}         // 模板类的成员函数
```

如上例所示，在类模板与其成员函数的定义中，带下划线的部分必须完全一致。

由于成员函数定义于类的结构之外，因此在实际定义时还要加上作用域运算符，用来指明其所属的类的名称。由于该类是一个类模板，因此在指定作用域运算符时必须要加上数据类型参数以分辨成员函数将来是属于此类模板产生的哪一种数据类型的模板类，如下面语句中的下划线部分的内容所示：

```
template <class Type> inline Vector <Type> :: Vector (const int s) {...}
```

在成员函数定义的最后是函数名、形参列表和函数体等和一般函数相同的定义部分。上述复杂的定义方式使得在程序的运行过程中，当数据类型参数被确定为一个具体的数据类型并生成了对应这个数据类型的对象后，就可以按照一般的数据类型那样来访问类的成员变量和成员函数。

在程序例15.7中，类模板的实际数据类型是在生成对象的时候确定到每一个具体的对象上的。例如：

```
Vector <int>      iVector(num);        // 整数类型对象
Vector <double>   dVector(num);        // 双精度浮点数类型对象
Vector <char>     cVector(num);        // 字符类型对象
```

在程序中，利用循环语句将数组所需要的数据一一读入，这里特别需要注意中间的执行语句：

```
iVector[i] = i;
dVector[i] = i;
```

编译器在识别程序的时候，按照从左向右的顺序进行扫描。首先扫描到iVector[i]，由于对象iVector所对应的类中有“[ ]”运算符的重载函数，因此编译器会将iVector[i]解释成iVector. operator [ ](i)。之后编译器会继续扫描，当扫描到符号“=”时，虽然对象iVector所对应的类中也有“=”运算符重载函数，但这时的“=”前面并不是一个iVector对象，因此“=”只能被解释成普通的等于号，整个语句也就被解释成：

```
( iVector.operator[ ](i) ) = i;
```

同理，浮点数类型的操作也是被解释成：

```
( dVector.operator[ ](i) ) = i;
```

此外，由于运算符“[ ]”的重载函数位于等于号的左方，被解释以后的语句如上所示，这就意味着要将变量i的值赋值给重载函数的返回值。如果重载函数的返回值的类型是一个普通变量类型，则返回的值代表一个数值常量，数值常量不可修改，上述语句不能编译运行。因此重载函数的返回值必须是一个左值（Lvalue，即可以被赋值的变量或者对象），所以该运算符重载函数的返回值数据类型被定义为引用类型。这样才可以将i的值赋给函数返回的引用变量所对应的内存空间，从而使得数组元素获得数值。

在程序例15.7中，真正会调用运算符“=”的重载函数的语句是：

```
cVector = 'A';
```

由于这时“=”前面是一个cVector对象，并且该对象所对应的类中存在运算符“=”的重载函数，因此编译器将语句解释成：

```
cVector.operator=( 'A');
```

这样，程序会执行“=”操作符重载函数中的循环语句，将cVector对象中的所有元素都赋值为字符'A'。最后，程序Li1507.cpp的运行结果如下：

```
请输入需要建立的数组的元素个数: 5
   iVector        dVector     cVector
         0       0.000000     A
         1        1.00000     A
         2        2.00000     A
         3        3.00000     A
         4        4.00000     A
```

### 15.3.2 类模板的友元函数

在前面的章节中我们学习了类的定义和使用，关于类的一个重要的应用就是友元函数。类模板与一般类相同，其中也可以定义友元函数，并且定义的方法和注意事项也和一般类相同。但是由于类模板毕竟是一种数据类型可以变化的类，它的友元函数也自然会存在与数据类型参数相关联的特殊写法，因此，我们根据是否使用了数据类型参数，将类模板的友元函数分为一般友元函数和特殊友元函数。所谓一般友元函数是指该友元函数没有使用任何类模板的数据类型参数列表中所定义的数据类型参数，即没有使用动态的数据类型参数的友元函数。例如：

```
template  <class Type>  class  Array  {
friend  void  display(int size);       // 一般友元函数
...
}
```

在上面的友元函数的定义中，没有使用类模板的数据类型参数Type，所以该友元函数是一般友元函数。

类模板中除了可以使用一般友元函数外，还可以使用两种特殊形式的友元函数。这两种特殊的友元函数都或多或少地使用了模板类的数据类型参数，但是由于使用的程度不同，因此带来了使用效果的不同。具体定义如下：

（1）绑定的类模板的友元函数（bounded class template friend function）。这种友元函数使用的数据类型参数与其所属的类的数据类型参数完全相同，好像友元函数的可变数据类型被其所属的类的可变数据类型所束缚了。

这种友元函数的定义格式如下：

```
template  < class Type >  class  <ClassName>
{
   friend  void  <FunctionName> (<ClassName> <Type> <param1>, ...);
   friend  Type  <FunctionName> (<ClassName> <Type> <param1>, int < param2>, ...);
   ...
};
```

在上面的友元函数中只使用了类模板的Type数据类型参数和int这样的基本数据类型，可以认为绑定的类模板的友元函数就是使用了数据类型参数的友元一般函数。

在使用绑定的类模板的友元函数的程序中，对于每一个由类模板产生的模板类来说，都有一个具有对应数据类型的友元函数。同时，每个由模板类生成的对象也都可以使用对应的友元函数来完成操作。假设我们为程序例15.7中定义的类模板按照如下方式扩充友元函数：

```
template  < class Type >  class  Vector
{
   friend  void  display(Vector <Type> array);
   friend  Type  get(Vector <Type> array, int index);
   ...
};
```

然后，同样生成三个不同数据类型的类对象:

```
Vector <int>     iVector(num);          // 整数类型对象
Vector <double>  dVector(num);          // 双精度浮点数类型对象
Vector <char>    cVector(num);          // 字符类型对象
```

其中，int类型的类对象会产生所有数据类型参数为int的友元函数，如下所示:

```
Vector <int>     iVector(num);
friend  void  display( Vector <int> array);
friend  int   get( Vector <int>array,  int index);
```

同理，double数据类型与char数据类型的类对象也有与自己的数据类型相同的友元函数:

```
Vector <double>    dVector(num);
friend  void  display(const Vector <double> array);
friend  int   get(const Vector <double> array, const int index);

Vector <char>   cVector(num);
friend  void  display(const Vector <char> array);
friend  int   get(const Vector <char> array, const int index);
```

(2) 非绑定的类模板的友元函数（unbounded class template friend function)。这种友元函数使用的数据类型参数与其所属的类的数据类型参数不完全相同，友元函数可以定义自己的数据类型参数，其使用起来要比上一种友元函数更加灵活和自由。

这种友元函数的定义格式如下:

```
template  < class Type >  class  <ClassName>
{
   template  <class T>  friend  void  <FunctionName> (<ClassName> <T> <param1>, ...);
   template  <class T>
   friend  Type  <FunctionName> (<ClassName> <T> <param1>, int < param2>, ...);
   ...
};
```

在上面的友元函数中没有使用类模板的Type数据类型参数，而是使用了自己定义的一个新的数据类型参数T。可以认为非绑定的类模板的友元函数就是在类模板中定义的友元函数模板。

这种友元函数由于对使用的灵活度要求较高，因此定义起来也要比前一种绑定的模板类的友元函数复杂。这种友元函数本身就是一个函数模板，因此在定义时函数的最前面要增加template关键字和友元函数自身的数据类型参数列表。这是非绑定的模板类的友元函数的主要特征。

对于程序例15.7中定义的类模板，我们同样按照非绑定的类模板的友元函数方式扩充友元函数，则扩充的效果如下所示:

```
template  < class Type >  class  Vector
{
   template  <class T>  friend  void  display(Vector <T> array);
   template  <class T>  friend  Type  get(Vector <T> array, int index);
   ...
};
```

则在display( )函数中，T是友元函数自己定义的数据类型参数，而不是像前一种友元函数那样，必须要使用所属的类模板定义的数据类型参数。这种情况相当于将类模板与函数模板相结合，在定义类模板的友元函数时使用了函数模板的技术。这种类模板的某个确定数据类型的对象通过使用友元函数，既可以处理本身数据类型的数据，也可以处理非本身数据类型的数据。可以形成对象的友元函数和所处理数据的数据类型之间的一对多的关系。例如:

```
Vector <int>     iVector(num);        // 整数类型对象
Vector <double>  dVector(num);        // 双精度浮点数类型对象
Vector <char>    cVector(num);        // 字符类型对象
```

其中iVector对象的友元函数display可以:

```
display(iVector);                     // 处理本身数据类型的数据
display(dVector);                     // 处理double类型的数据
display(cVector);                     // 处理char类型的数据
```

这时，iVector对象的友元函数display与各对象之间的关系就是一个一对多的关系。同样，dVector和cVector对象的友元函数display也同样可以实现上述的一对多关系。下面在程序例15.8中综合说明类模板的友元函数的定义和使用。

**例15.8**　演示类模板的友元函数的使用。

```
#include  <iostream.h>
#include  <iomanip.h>
template  < class Type >  class  Vector
{
   friend  ostream&  operator<<(ostream& os, Vector <Type> &array);
private:
   Type*  data;
   int    size;
public:
   Vector(int s);
   ~Vector( );
   Type&  operator [ ] (const int index);
   void  operator = (Type value);
};
template  <class Type>  inline  Vector <Type>  ::  Vector(int s)
{
   size = s;
   data = new Type[size];
   for(int i=0; i<size; i++)
       data[i] = 0;
}
template  <class Type>  inline  Vector  <Type>  ::  ~Vector( )
{
   delete  [ ] data;
}
template  <class Type>  Type&  Vector  <Type> :: operator[ ](const int index)
{
   return  data[index];
}
template  <class Type>  void  Vector  <Type>  ::  operator = (Type value)
{
   for(int i=0; i<size; i++)
       data[i] = value;
}
template  <class Type>
ostream&  operator << (ostream& os, Vector <Type> &array)
{
   for(int i=0; i<array.size; i++)
```

```
        os << array.data[i] << endl;
    return  os;
}
void  main( )
{
    int num;
    cout << "请输入需要建立的数组的元素个数: ";
    cin >> num;
    // 生成类的对象
    Vector <int>     iVector(num);              // 整数类型对象
    Vector <double>  dVector(num);              // 双精度浮点数类型对象
    Vector <char>    cVector(num);              // 字符类型对象
    iVector = 100;
    dVector = 7.56;
    cVector = 'A';
    cout.setf(ios::showpoint);                  // 显示浮点数的小数部分
    cout << "The integer type class:" << endl << iVector << endl;
    cout << "The double floating type class:" << endl << dVector << endl;
    cout << "The character type class:" << endl << cVector << endl;
}
```

在程序例15.8中的类模板Vector中定义了一个绑定的模板类的友元函数:

```
friend  ostream&  operator << (ostream& os, Vector <Type> &array);
```

该函数使用了运算符重载技术重载了运算符“<<”，使得类模板Vector可以用于各种数据类型对象的数据输出。

### 15.3.3 类模板的特殊处理

与函数模板一样，模板类有时也必须要对某一种数据类型做特别的处理。这时可以像15.2.2节中的模板函数的处理方法一样，定义一个用于处理特殊情况的模板类。在定义这个特别类的时候，只要将数据类型参数定义为所要特别处理的数据类型就行了。如果类模板中只是某个成员函数无法适应某个数据类型，则不需要对类模板本身做特殊处理，只要对该类模板重新定义一个处理这个特殊情况的成员函数即可。这时，其实是对类模板中的成员函数做重载运算。

例如，若例15.7中的类模板Vector中的赋值运算符成员函数对于字符串类型（char *）的处理无法达到要求，则程序设计者可以重新定义一个新的进行特殊处理的成员函数来处理字符串类型的数据:

```
template  <char *>  void  Vector <char *> :: operator =(char *value)
{
    // 重新定义函数体的实际操作部分
    ...
}
```

也可以对整个模板类重新定义一个特别的类，使得这个特殊类型的类有一个完整的定义。例如:

```
class  Vector  <char *>
{
   friend  ostream&  operator << (ostream& os, Array <char *> &array);
private:
   char**  data;
   int    size;
public:
   Vector(const int size);
   ~Vector( );
```

```
    char&  operator [ ] (const int index);
    void   operator = (char *temp);
};
```

在这个特别类的定义中，类的起始部分不再使用关键字template和数据类型参数列表，但是类名称的后面必须加上已经定义的数据类型，以表明该类是专门用来处理某种数据类型的特别类。这种特殊类模板的定义方法称为直接实例化（Explicit Instantiation）。此外，要必须注意的是，虽然特别类的定义不必与原来类模板中的定义完全相同，但是一旦定义了处理某种数据类型的特殊类，则将来所有属于该数据类型的对象的数据成员和成员函数的定义和使用便都由该特别类负责，而不可以使用任何由类模板产生的模板类中的数据成员和成员函数。这也和前面的函数模板的特殊处理情况类似，特别类的优先级高于类模板，系统先对特别类进行匹配，一旦找到了符合条件的特别类，就不会再对类模板进行匹配和类型转换操作了。例如:

```
template  < class Type >  class  Test
{
private:
      Type  val;
public:
      Test(Type  var);
      friend  ostream&  operator << (ostream& os, Type &value);
};
// 类模板的特别类，用以处理char数据类型
class  Test  <char>
{
private:
      char  val;
public:
      Test(char  var);
      void  operator = (char var);
};
void  main( )
{
    Test<int>     T1(30);          // 生成整数类型对象
    Test<char>    T2('C');         // 生成字符类型对象
    cout << T1;
    cout << T2;                    // 错误
}
```

由于对象T2的数据类型是字符类型，因此是由特别类Test<char>来定义的，而该类中没有重载定义“<<”运算符，所以想要通过输出运算符函数来将对象输出到屏幕是错误的。而T1对象的数据类型是整数类型，是由一般Test类模板所定义的，并且该类模板中重载定义了“<<”运算符，因此可以直接通过输出运算符函数来将对象输出到屏幕上。

## 15.4 总结

通过本章的学习，我们可以学习和掌握函数模板和类模板的各种定义和使用方法。可以利用模板的概念来解决大量需要用相同的算法来处理不同数据类型数据的情况，还可以在一般的函数模板和类模板无法处理某些特别的数据类型时，通过定义特别的函数和类的方法来进行特别处理。

由于函数模板和类模板使用方便、灵活，因此在目前的面向对象程序设计中已经占据了重要的地位。在Microsoft公司的Visual C++的核心类库MFC中专门有一组类使用了模板技术，它们统称为基本模板类

(Template-Based Classes)。这些类在进行对数组、链表等用来存放各种不同数据类型数据的基本数据结构的操作时有着十分重要的作用。

甚至有的编译器希望程序员在编写C++程序时尽量少用或不用宏，而代之以模板，以便能够充分发挥C++的特性。

因此，熟悉和掌握模板的编程技术，对于使用C++进行Windows开发等高级编程工作是十分有益的。

## 习题15

1. 使用模板技术，可以减轻使用重载函数造成的书写和维护的重复工作量，那么是否在C++中就可以取消函数重载这一技术呢？为什么？
2. 请编写程序，使用模板函数技术实现计算数值幂次方的功能。要求程序按照浮点数方式接受用户输入的底数和幂数两个数值到两个变量中，然后按照如下方式调用函数，求出$x^y$的值。

```
z = power(x, y);
```

分别让输入的变量中的数据按照整数格式和浮点数格式计算出的$x^y$的值，并且输出到屏幕上。

3. 请编写程序，使用模板函数技术对不同类型的数组进行求和计算，要求模板函数具有两个参数分别接收数组的首地址和数组中元素的个数。
4. 请编写程序，构造一个模板类CValue，这个类中可以存放5个数值，并且具有两个成员函数。成员函数的功能分别是求出5个数中的最大值以及5个数中的最小值。用户输入随机的5个数，将最大值和最小值输出在屏幕上。
5. 请编写程序，按照第3题的要求构造模板类CValue，但是将Max( )和Min( )改造成友元函数，使用友元函数输出多个不同类型对象的最大值和最小值。
6. 请编写程序，构造一个模板类CArray，这个类中有两个成员变量：一个为指针变量，用来指向数组元素所在空间的首地址，数组元素可以是任意数据类型。另一个为数组元素的个数。请定义类的成员函数，实现数组元素的赋值，数组元素的输出，累加和的计算和输出，平均数的计算和输出，最大值和最小值的求解和输出。

# 附录A ASCII码表

| ASCII值 | 控制字符 | ASCII值 | 字符 | ASCII值 | 字符 | ASCII值 | 字符 |
|---|---|---|---|---|---|---|---|
| 0 | NUL | 32 | | 64 | @ | 96 | ` |
| 1 | SOH | 33 | ! | 65 | A | 97 | a |
| 2 | STX | 34 | " | 66 | B | 98 | b |
| 3 | ETX | 35 | # | 67 | C | 99 | c |
| 4 | EOT | 36 | $ | 68 | D | 100 | d |
| 5 | END | 37 | % | 69 | E | 101 | e |
| 6 | ACK | 38 | & | 70 | F | 102 | f |
| 7 | BEL | 39 | ' | 71 | G | 103 | g |
| 8 | BS | 40 | ( | 72 | H | 104 | h |
| 9 | HT | 41 | ) | 73 | I | 105 | i |
| 10 | LF | 42 | * | 74 | J | 106 | j |
| 11 | VT | 43 | + | 75 | K | 107 | k |
| 12 | FF | 44 | , | 76 | L | 108 | l |
| 13 | CR | 45 | - | 77 | M | 109 | m |
| 14 | SO | 46 | . | 78 | N | 110 | n |
| 15 | SI | 47 | / | 79 | O | 111 | o |
| 16 | DEL | 48 | 0 | 80 | P | 112 | p |
| 17 | DC1 | 49 | 1 | 81 | Q | 113 | q |
| 18 | DC2 | 50 | 2 | 82 | R | 114 | r |
| 19 | DC3 | 51 | 3 | 83 | S | 115 | s |
| 20 | DC4 | 52 | 4 | 84 | T | 116 | t |
| 21 | NAK | 53 | 5 | 85 | U | 117 | u |
| 22 | SYN | 54 | 6 | 86 | V | 118 | v |
| 23 | ETB | 55 | 7 | 87 | W | 119 | w |
| 24 | CAN | 56 | 8 | 88 | X | 120 | x |
| 25 | EM | 57 | 9 | 89 | Y | 121 | y |
| 26 | SUB | 58 | : | 90 | Z | 122 | z |
| 27 | ESC | 59 | ; | 91 | [ | 123 | { |
| 28 | FS | 60 | < | 92 | \ | 124 | \| |
| 29 | GS | 61 | = | 93 | ] | 125 | } |
| 30 | RS | 62 | > | 94 | ^ | 126 | ~ |
| 31 | US | 63 | ? | 95 | _ | 127 | △ |

# 附录B 常用库函数

库函数并不是C++语言的组成部分。它是由人们根据需要编制并提供给用户使用的。每一种C++编译系统都提供了一批库函数，不同的编译系统所提供的库函数的数目和函数名以及函数功能并不完全相同。

但为了方便用户使用库函数，VC++编译器提供了大量的库函数。用到库函数时必须包含相应的头文件。需了解VC++提供的库函数请查阅有关的手册。本附录列出了一些最常用的库函数。

## 1. 常用数学函数

用到以下函数时，要包含头文件math.h。

| 函数原型 | 功 能 | 返回值 | 说 明 |
|---|---|---|---|
| int abs(int x) | 求整数的绝对值 | 绝对值 | |
| double acos(double x) | arcos(x) | 计算结果 | $-1 \leqslant x \leqslant 1$ |
| double asin(double x) | arcsin(x) | 计算结果 | $-1 \leqslant x \leqslant 1$ |
| double atan(double x) | arctan(x) | 计算结果 | |
| double cos(double x) | cos(x) | 计算结果 | 弧度 |
| double cosh(double x) | 双曲余弦 | 计算结果 | |
| double exp(double x) | 求$e^x$ | 计算结果 | |
| double fabs(double x) | 求实数的绝对值 | 绝对值 | |
| double fmod(double x,double y) | 求x/y的余数 | 余数 | |
| long labs(long x) | 求长整型数的绝对值 | 绝对值 | |
| double log(double x) | ln(x) | 计算结果 | |
| double log10(double x) | 求以10为底的对数 | 计算结果 | |
| double pow(double x,double y) | 求$x^y$ | 计算结果 | |
| double sin(double x) | sin(x) | 计算结果 | |
| double sqrt(double x) | 求平方根 | 计算结果 | |
| double tan(double x) | 正切函数 | 计算结果 | |
| double modf(double x,double *y) | 取x的整数部分放到y所指向的单元中 | x的小数部分 | |

## 2. 字符串处理函数

用到以下字符串处理函数时，要包含头文件string.h。

| 函数原型 | 功 能 | 返回值 | 说 明 |
|---|---|---|---|
| void * memcpy(void * p1,const void *p2,size_t n) | 存储器拷贝，将p2所指向的n个字节拷贝到p1所指向的存储区中 | 目的存储区的起始地址 | 实现任意数据类型之间的拷贝 |
| void * memset(void * p,int v,size_t n) | 将v的值作为p所指向的区域的值，n是p所指向区域的大小 | 返回该区域的起始地址 | |
| char * strcpy(char * p1,const char *p2) | 将p2所指向的字符串拷贝到p1所指向的区域中 | 目的存储区的起始地址 | |

（续）

| 函数原型 | 功　能 | 返回值 | 说　明 |
| --- | --- | --- | --- |
| char * strcat(char * p1,const char *p2) | 将p2所指向的字符串接到p1所指向的字符串的后面 | 目的存储区的起始地址 | |
| int * strcmp(const char * p1,const char *p2) | 两个字符串比较 | 两字符串相同，返回0；若p1所指向的字符串小于p2所指向的字符串，返回负数；否则，返回正数 | |
| size_t strlen(const char *) | 求字符串的长度 | 字符串中包含的字符个数 | |
| char * strncat(char * p1,const char *p2,size_t n) | 将p2所指向的字符串（最多n个字符）接到p1所指向的字符串的后面 | 目的存储区的起始地址 | |
| int * strncmp(char * p1,const char *p2,size_t n) | 前两个参数与函数strcmp( )相同，最多比较n个字符 | 与函数strcmp( )相同 | |
| char * strncpy(char * p1,const char *p2,size_t n) | 前两个参数与函数strcpy( )相同，最多拷贝n个字符 | 与函数strcpy( )相同 | |
| char * strstr(char * p1,const char *p2) | p2所指向的字符串是否是p1所指向的字符串的子串 | 若是子串，返回子串开始位置；否则返回0 | |

### 3. 常用的其他函数

用到以下字符串处理函数时，要包含头文件stdlib.h。

| 函数原型 | 功　能 | 返回值 | 说　明 |
| --- | --- | --- | --- |
| void abort(void) | 终止程序执行 | | 不做结束工作 |
| void exit(int) | 终止程序执行 | | 做结束工作 |
| int abs(int) | 求整数的绝对值 | 返回绝对值 | |
| long labs(long) | 求长整数的绝对值 | 返回绝对值 | |
| double atof(const char *s) | 将s所指向的字符串转换成实数 | 返回实数值 | |
| int atoi(const char *) | 将字符串转换成整数 | 返回整数值 | |
| long atol(const char *) | 将字符串转换成长整数 | 返回长整数值 | |
| int rand(void) | 产生一个随机数 | 返回随机数 | |
| void srand(unsigned int) | 初始化随机数发生器 | | |
| int system(const char *s) | 将s所指向的字符串作为一个可执行文件，并执行之 | | |
| max(a,b) | 求两个数中的大数 | 返回大数 | 参数可为任意类型 |
| min(a,b) | 求两个数中的小数 | 返回小数 | 参数可为任意类型 |

### 4. 实现键盘和文件输入／输出的成员函数

用到以下函数时，要包含头文件iostream.h。

| 函数原型 | 功 能 | 返回值 | 说 明 |
|---|---|---|---|
| cin>>v | 输入值送给变量v | | |
| cout<<exp | 输出表达式exp的值 | | |
| istream & istream::get(char &c) | 输入字符送给变量c | | |
| istream & istream::get(char *,int, char='\n') | 输入一行字符串 | | |
| istream & istream::getline(char *,int, char='\n') | 输入一行字符串 | | |
| void ifstream ::open(const char *,int= ios::in,int=filebuf:: openprot) | 打开输入文件 | | |
| void ofstream ::open(const char *,int=ios::out,int=filebuf:: openprot) | 打开输出文件 | | |
| void fstream ::open(const char *, int,int= filebuf::openprot) | 打开输入/输出文件 | | |
| ifstream :: ifstream(const char *, int=ios::in, int=filebuf:: openprot) | 构造函数打开输入文件 | | |
| ofstream :: ofstream(const char *, int= ios::out, int=filebuf:: openprot) | 构造函数打开输出文件 | | |
| fstream :: fstream(const char *, int, int =filebuf::openprot) | 构造函数打开输入/输出文件 | | |
| void ifstream::close( ) | 关闭输入文件 | | |
| void ofstream::close( ) | 关闭输出文件 | | |
| void fstream::close( ) | 关闭输入/输出文件 | | |
| istream &istream::read(char *,int) | 从文件中读取数据 | | |
| ostream & ostream::write(const char *,int) | 将数据写入文件中 | | |
| int ios::eof( ) | 是否到达打开的文件的尾部 | 1为到达文件尾部，0为没有到达文件尾部 | |
| istream &istream::seekg(streampos) | 移动输入文件的指针 | | |
| istream &istream::seekg(stream-off,ios::seek_dir) | 移动输入文件的指针 | | |
| streampos istream::tellg( ) | 取输入文件的指针 | | |
| ostream &ostream::seekp(streampos) | 移动输出文件的指针 | | |
| ostream &ostream::seekp(stream-off,ios::seek_dir) | 移动输出文件的指针 | | |
| streampos ostream::tellp( ) | 取输出文件的指针 | | |

# 参考文献


1 张岳新编著. Visual C++程序设计. 苏州：苏州大学出版社，2002

2 钱能主编. C++程序设计教程. 北京：清华大学出版社，1999

3 吕凤翥著. C++语言基础教程. 北京：清华大学出版社，2000

4 谭浩强著. C程序设计（第二版）. 北京：清华大学出版社，1999

5 潘金贵，潇柔，吴卫华等编著. Borland C++程序设计教程. 南京：南京大学出版社，1994

6 宛延闿编著. C++语言和面向对象程序设计. 北京：清华大学出版社，1998

7 张基温编著. C++程序设计基础. 北京：高等教育出版社，1996

8 李智慧，秦成编著. C++ Builder 4.0 从入门到精通. 北京：清华大学出版社，1999

9 蔡明志编著. Borland C++ 3.1 软件集成技术与范例. 北京：海洋出版社，1993

10 （美）D. S. Malik 著. C++编程——数据结构与程序设计方法学. 晏海华，蔡旭辉，常鸿等译. 电子工业出版社，2003

11 （英）John R. Hubbard 著. C++编程习题与解答. 徐漫江，王栋，何路，等译. 北京：机械工业出版社，2002